US006643736B1

(12) **United States Patent**
Chen

(10) **Patent No.: US 6,643,736 B1**
(45) **Date of Patent: Nov. 4, 2003**

(54) **SCRATCH PAD MEMORIES**

(75) Inventor: **Hong-Yi Hubert Chen**, Fremont, CA (US)

(73) Assignee: **Arm Limited**, Cambridge (GB)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 268 days.

(21) Appl. No.: **09/650,244**

(22) Filed: **Aug. 29, 2000**

(51) **Int. Cl.**^{7} **G06F 12/08**

(52) **U.S. Cl.** **711/117**; 711/122; 711/123; 712/205

(58) **Field of Search** 711/117, 3, 119, 711/122, 132, 120, 123, 173; 345/503, 551, 557; 713/500; 712/204–207

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,843,542 A	*	6/1989	Dashiell et al.	711/119
5,067,078 A	*	11/1991	Talgam et al.	711/3
5,553,276 A	*	9/1996	Dean	713/500
5,893,159 A	*	4/1999	Schneider	711/150
5,922,066 A	*	7/1999	Cho et al.	712/204
5,966,734 A	*	10/1999	Mohamed et al.	711/173

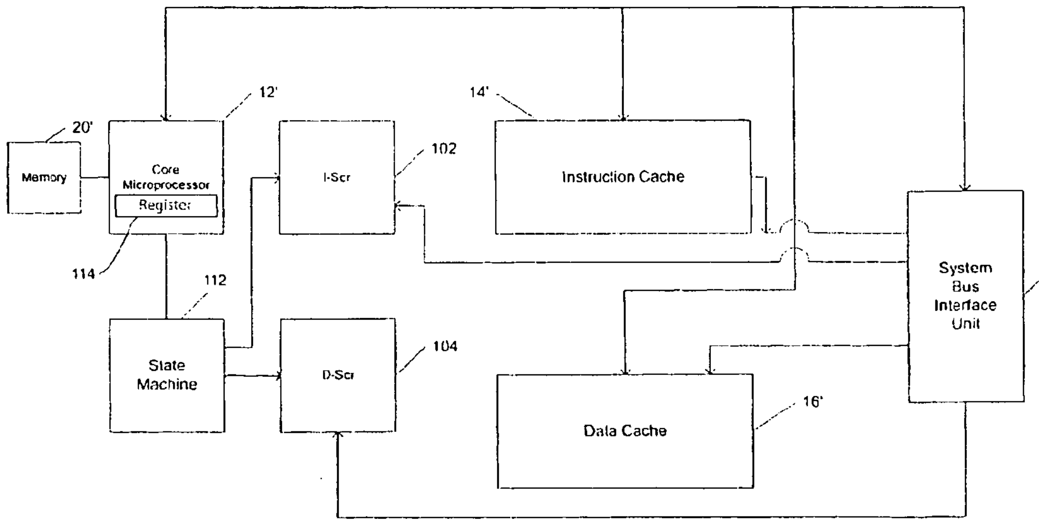
* cited by examiner

Primary Examiner—Nasser Moazzami
(74) *Attorney, Agent, or Firm*—Nixon & Vanderhye, P.C.

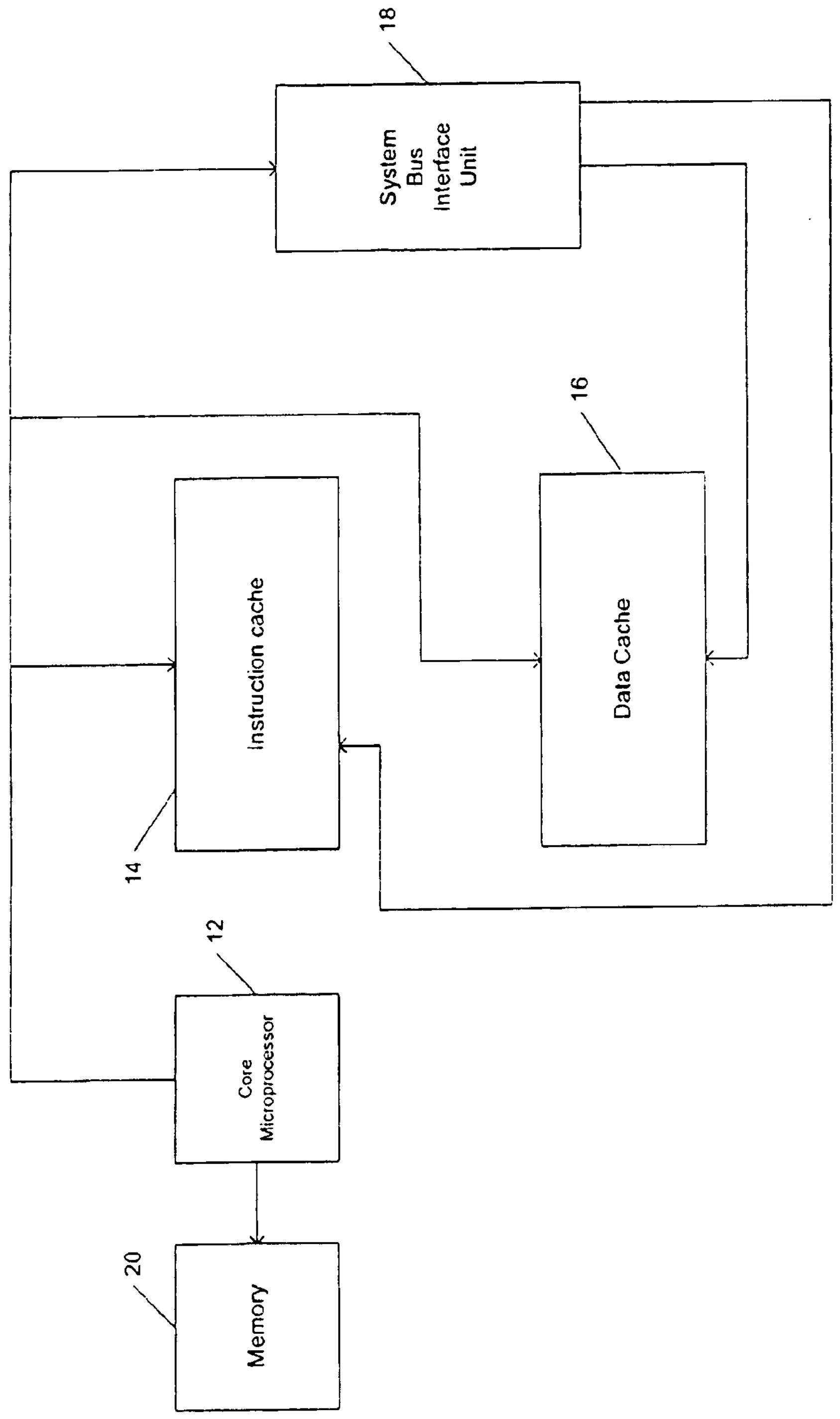
(57) **ABSTRACT**

A processing system is disclosed. The processing system includes at least one cache and at least one scratch pad memory. The system also includes a processor for accessing the at least one cache and at least one scratch pad memory. The at least one scratch pad memory is smaller in size than the at least one cache. The processor accesses the data in the at least one scratch pad memory before accessing the at least one cache to determine if the appropriate data is therein. There are two important features of the present invention. The first feature is that an instruction can be utilized to fill a scratch pad memory with the appropriate data in an efficient manner. The second feature is that once the scratch pad has the appropriate data, it can be accessed more efficiently to retrieve this data within the cache and memory space not needed for this data. This has a particular advantage for frequently used routines, such as a mathematical algorithm to minimize the amount of space utilized in the cache for such routines. Accordingly, the complexity of the cache is not required using the scratch pad memory as well as space within the cache is not utilized.

3 Claims, 3 Drawing Sheets



100



10

Figure 1 (Prior Art)

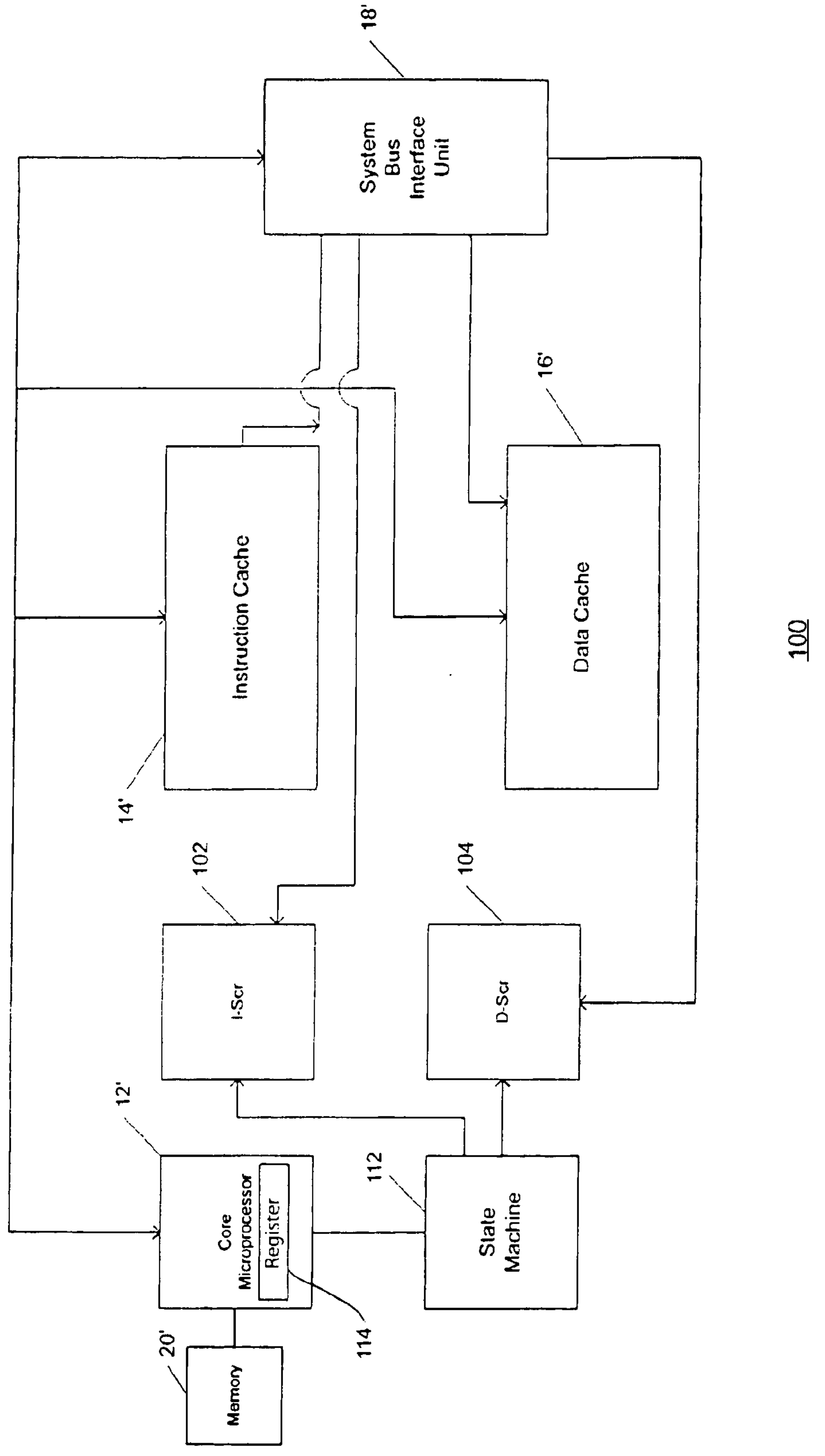

18'
System Bus Interface Unit
16'
Instruction Cache
Data Cache
14'
100
102
104
I-Scr
D-Scr
12'
Core Microprocessor
Register
112
State Machine
114
20'
Memory

Figure 2

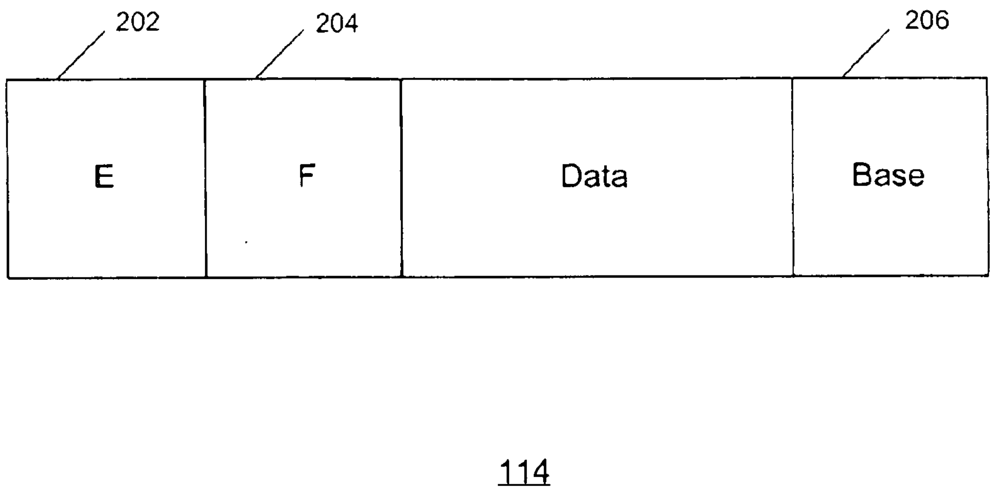


Figure 3

SCRATCH PAD MEMORIES

FIELD OF THE INVENTION

The present invention relates generally to a processing system and more particularly to a processing system that includes a scratch pad for improved performance.

BACKGROUND OF THE INVENTION

Processor architectures are utilized for a variety of functions. FIG. 1 is a simple block diagram of a conventional processing system **10**. The processing system **10** includes a core processor **12** which controls a system bus interface unit **18**. The core processor **12** also interacts with an instruction cache and a data cache. Typically, the core processor retrieves information from the data cache or the instructions for operation rather than obtaining data from system memory as is well known. Since the data cache and instruction cache are smaller in size, data can be accessed from them more readily if it is resident therein.

In this type of processing system, oftentimes small routines are provided which can further affect the performance of the system. Accordingly, the caches are placed therein to is allow faster access rather than having to access system memory. Although these caches are faster than system memory, they still are relatively slow if the routine needs to be accessed on a continual basis therefrom. For example, small routines may take up several cycles which can become a performance bottleneck in a processing system. So what is desired is a system which will allow one to more quickly access and obtain certain routines and therefore improve the overall performance of the system in the data cache without wasting memory space.

The system must be easy to implement utilizing existing technologies. The present invention addresses such a need.

SUMMARY OF THE INVENTION

A processing system is disclosed. The processing system includes at least one cache and at least one scratch pad memory. The system also includes a processor for accessing the at least one cache and at least one scratch pad memory. The at least one scratch pad memory is smaller in size than the at least one cache. The processor accesses the data in the at least one scratch pad memory before accessing the at least one cache to determine if the appropriate data is therein.

There are two important features of the present invention. The first feature is that an instruction can be utilized to fill a scratch pad memory with the appropriate data in an efficient manner. The second feature is that once the scratch pad has the appropriate data, it can be accessed more efficiently to retrieve this data within the cache and memory space not needed for this data. This has a particular advantage for frequently used routines, such as a mathematical algorithm to minimize the amount of space utilized in the cache for such routines. Accordingly, the complexity of the cache is not required using the scratch pad memory as well as space within the cache is not utilized.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a simple block diagram of a conventional processing system.

FIG. 2 is a simple block diagram of a system in accordance with the present invention.

FIG. 3 is a diagram of a register utilized for a scratch pad in accordance with the present invention.

DETAILED DESCRIPTION

The present invention relates generally to a processing system and more particularly to a processing system that includes a scratch pad for improved performance. The following description is presented to enable one of ordinary skill in the art to make and use the invention and is provided in the context of a patent application and its requirements. Various modifications to the preferred embodiment and the generic principles and features described herein will be readily apparent to those skilled in the art. Thus, the present invention is not intended to be limited to the embodiment shown but is to be accorded the widest scope consistent with the principles and features described herein.

FIG. 2 is a block diagram of a system **100** in accordance with the present invention. Those elements that are similar to those of FIG. 1 are given similar reference numbers. As is seen, a scratch pad memory **102** is provided for the instruction cache and a scratch pad memory **104** is provided with the data cache **16'**. The scratch pad memories **102** and **104** are typically 2 Kb in size as compared to a 8 Kb data cache and 8 Kb instruction cache. In a preferred embodiment, the scratch pad memories **102** and **104** have the highest priority when accessing data. A state machine **112** is coupled to the instruction cache **14'** and data cache **16'** and interacts with scratch pad memories **102**, **104** and a register **114** with the core processor **12'**. The state machine **112** provides access to the register **114** within the core processor **12'**.

FIG. 3 is a diagram of the register **114** utilized for a scratch pad memory in accordance with the present invention. The register **114** includes an enable (E) bit **202** for enabling the scratch pad memory, a fill (F) bit **204** to fill the scratch pad memory and bits **206** for storing the base address for the instruction tat causes the filling of the scratch pad memory **102** and **104**.

There are two important features of the present invention. The first feature is that an instruction can be utilized to fill a scratch pad memory with the appropriate data in an efficient manner. The second feature is that once the scratch pad has the appropriate data, it can be accessed more efficiently to retrieve this data within the cache and memory space not needed for this data. This has a particular advantage for frequently used routines, such as a mathematical algorithm to minimize the amount of space utilized in the cache for such routines. Accordingly, the complexity of the cache is not required using the scratch pad memory as well as space within the cache is not utilized.

The operation of the present invention will be described in the context of the instruction cache **14'** and its associated scratch pad memory **102** but one of ordinary skill in the art recognizes that the data cache **16'** and its associated scratch pad memory **104** could be utilized in a similar manner.

A system in accordance with the present invention operates in the following manner. First the filling of the scratch pads will be described. Assuming there is a cache miss, then the data from system memory will be read, and the scratch pad memory **102** will be filled. The scratch pad **102** will be filled based upon an instruction resident in the register **114**. In a preferred embodiment the enable bit is set to 1 and the fill bit is set to 1 to indicate that data can be loaded into the scratch pad memory. The core processor **12'** reads the data from the base address range of the register **114** and this will be the data that will be provided to the scratch pad memory **102**. The state machine **112** captures the event of writing into the register **114** and causes the system bus unit **18'** to fill the scratch pad memory **102**. When the scratch pad memory **102** is filled, the pipeline is released by the processor 12'. Therefore, the scratch pad memory **102** then includes the routine (for example, a mathematical algorithm). Once released, then processing can continue. scratch pad memory **106**. The state machine **112** captures the event of writing into the register **114** and causes the system bus unit 16' to fill the scratch pad memory **106**. When the scratch pad memory **106** is filled, the pipeline is released by the processor 12'. Therefore, the scratch pad memory **106** then includes the routine (for example, a mathematical algorithm). Once released, then processing can continue.

Next, the accessing of the scratch pad memory **102** will be described. Accordingly, when the particular routine needs to be accessed, first the processor 12' accesses the scratch pad memory **102** to determine whether the data is there. If the data is there, it can be read directly from the scratch pad in a more efficient manner than reading it from the data cache. This can be performed several times to allow the processor to allow for faster access to the data. If the data is not there then the processor accesses the data in the cache. If the data is not within the scratch pad memory or the data cache then the processor will obtain the data from system memory 20'.

Accordingly, through a system and method in accordance with the present invention a processing system's performance is significantly improved since data can be accessed more quickly from the scratch pad memory. In addition, the filling of the scratch pad memory can be accomplished in a simple and straightforward manner.

Although the present invention has been described in accordance with the embodiments shown, one of ordinary skill in the art will readily recognize that there could be

variations to the embodiments and those variations would be within the spirit and scope of the present invention. Accordingly, many modifications may be made by one of ordinary skill in the art without departing from the spirit and scope of the appended claims.

What is claimed is:

1. A system for improving the performance of a processing system, the processing system including a processor and at least one cache, the system comprising:

a scratch pad memory which can be accessed by the processor;

a mechanism for providing the scratch pad memory with the appropriate data when the data is not within the at least one cache, wherein the scratch pad is smaller than the at least one cache and is accessed by the processor before the at least one cache;

a register within the processor;

a state machine for accessing the register when the scratch pad memory is to be filled;

an instruction within the register for initiating the filling of the scratch pad memory; and

a system interface unit for filling the scratch pad memory with the appropriate data.

2. The system of claim **1** wherein the filling of the scratch pad is provided by a system memory.

3. The system of claim **2** wherein the register comprises an enable bit, a fill bit and a base address for the instruction.

* * * * *

US006085314A

United States Patent [19]

Asghar et al.

[11] **Patent Number:** **6,085,314**

[45] **Date of Patent:** ***Jul. 4, 2000**

[54] **CENTRAL PROCESSING UNIT INCLUDING APX AND DSP CORES AND INCLUDING SELECTABLE APX AND DSP EXECUTION MODES**

[75] Inventors: **Saf Asghar**, Austin, Tex.; **Andrew Mills**, Coto-de-Caza, Calif.

[73] Assignee: **Advnced Micro Devices, Inc.**, Sunnyvale, Calif.

[*] Notice: This patent is subject to a terminal disclaimer.

[21] Appl. No.: **08/969,858**

[22] Filed: **Nov. 14, 1997**

Related U.S. Application Data

[63] Continuation-in-part of application No. 08/618,243, Mar. 18, 1996, Pat. No. 5,794,068.

[51] **Int. Cl.**⁷ **G06F 9/30**; G06F 15/163

[52] **U.S. Cl.** **712/213**; 712/23; 712/35; 712/207

[58] **Field of Search** 395/800.23, 800.35, 395/383, 387, 359, 800.75; 712/23, 35, 204, 206, 213, 207, 203, 208, 210, 211, 217, 230, 245

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,173,041	10/1979	Dvorak et al. .	
5,355,485	10/1994	Denio et al. .	
5,542,059	7/1996	Blomgren .	
5,588,118	12/1996	Mandava et al. .	
5,619,665	4/1997	Emma .	
5,721,945	2/1998	Mills et al.	395/800.35

(List continued on next page.)

FOREIGN PATENT DOCUMENTS

0 071 028	2/1983	European Pat. Off. .
0 442 041 A2	8/1991	European Pat. Off. .
0 478 904 A2	4/1992	European Pat. Off. .
0465054	4/1996	European Pat. Off. .
3711651 A1	10/1988	Germany .
97/35252	9/1997	WIPO .

OTHER PUBLICATIONS

International Search Report for PCT/US 96/19586 dated Apr. 28, 1997.

Halfhill, T.R., "AMD K6 Takes on Intel P6," BYTE, vol. 21, No. 1, Jan. 1, 1996, pp. 67, 68, 70 & 72.

International Search Report for PCT/US 97/01067 mailed Jun. 2, 1997.

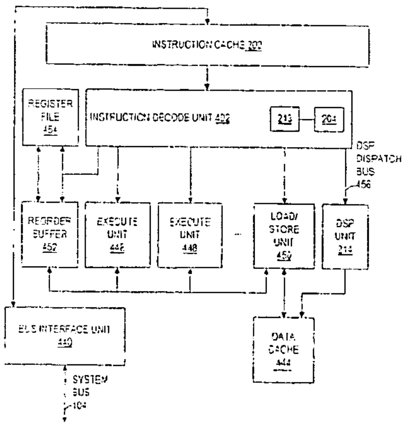
International Search Report for PCT/US 98/10175 dated Sep. 2, 1998.

Primary Examiner—Larry D. Donaghue
Attorney, Agent, or Firm—Conley, Rose & Tayon, P.C.; Jeffrey C. Hood; Eric A. Stephenson

[57] **ABSTRACT**

A CPU or microprocessor which includes a general purpose CPU component, such as an X86 core, and also includes a DSP core. In a first embodiment, the CPU receives general purpose instructions, such as X86 instructions, wherein certain X86 instruction sequences implement DSP functions. The CPU includes a processor mode register which is written with one or more processor mode bits to indicate whether an instruction sequence implements a DSP function. The CPU also includes an intelligent DSP function decoder or preprocessor which examines the processor mode bits and determines if a DSP function is being executed. If a DSP function is being implemented by an instruction sequence, the DSP function decoder converts or maps the opcodes to a DSP macro instruction that is provided to the DSP core. The DSP core executes one or more DSP instructions to implement the desired DSP function in response to the macro instruction. If the processor mode bits indicate that X86 instructions in the instruction memory do not implement a DSP-type function, the opcodes are provided to the X86 core as which occurs in current prior art computer systems. In a second embodiment, the CPU receives sequences of instructions comprising X86 instructions and DSP instructions. The processor mode register is written with one or more processor mode bits to indicate whether an instruction sequence comprises X86 or DSP instructions, and the instructions are routed to the X86 core or to the DSP core accordingly.

20 Claims, 11 Drawing Sheets



U.S. PATENT DOCUMENTS

5,740,420	4/1998	Palaniswami	395/595
5,768,553	6/1998	Tran	395/384
5,781,750	7/1998	Blomgren et al.	395/385
5,781,792	7/1998	Asghar et al.	395/800.35
5,784,640	7/1998	Asghar et al.	395/800.35
5,790,824	8/1998	Asghar et al.	395/385
5,794,068	8/1998	Asghar et al.	395/800.35
5,829,031	10/1998	Lynch	711/137
5,854,913	12/1998	Goetz et al.	395/386

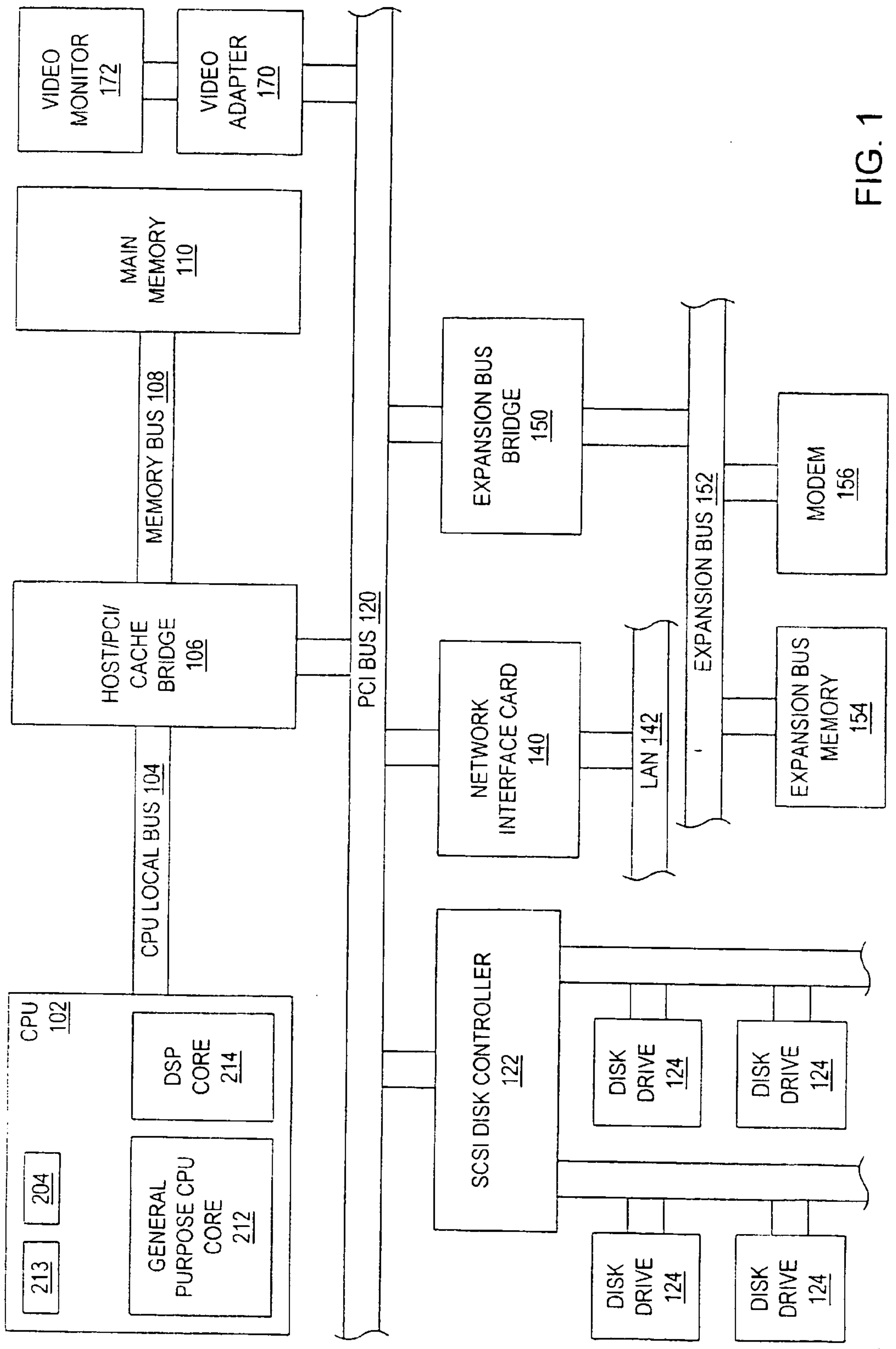

VIDEO MONITOR 172
VIDEO ADAPTER 170
MAIN MEMORY 110
MEMORY BUS 108
HOST/PCI/ CACHE BRIDGE 106
CPU LOCAL BUS 104
CPU 102
213
204
GENERAL PURPOSE CPU CORE 212
DSP CORE 214
PCI BUS 120
EXPANSION BUS BRIDGE 150
NETWORK INTERFACE CARD 140
LAN 142
EXPANSION BUS 152
MODEM 156
EXPANSION BUS MEMORY 154
SCSI DISK CONTROLLER 122
DISK DRIVE 124
DISK DRIVE 124
DISK DRIVE 124
DISK DRIVE 124

FIG. 1

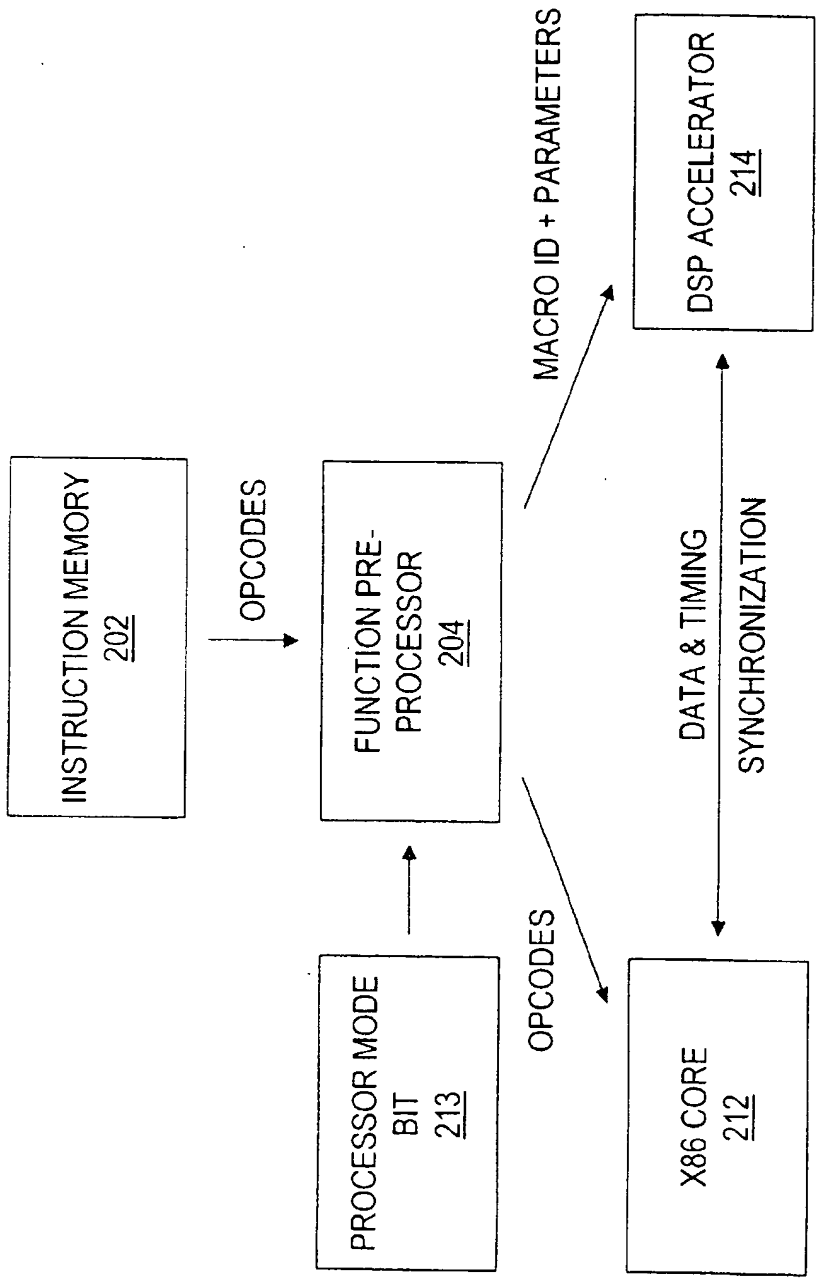

INSTRUCTION MEMORY
202
OPCODES
FUNCTION PRE-PROCESSOR
204
PROCESSOR MODE BIT
213
MACRO ID + PARAMETERS
OPCODES
DSP ACCELERATOR
214
DATA & TIMING SYNCHRONIZATION
X86 CORE
212

FIG. 2

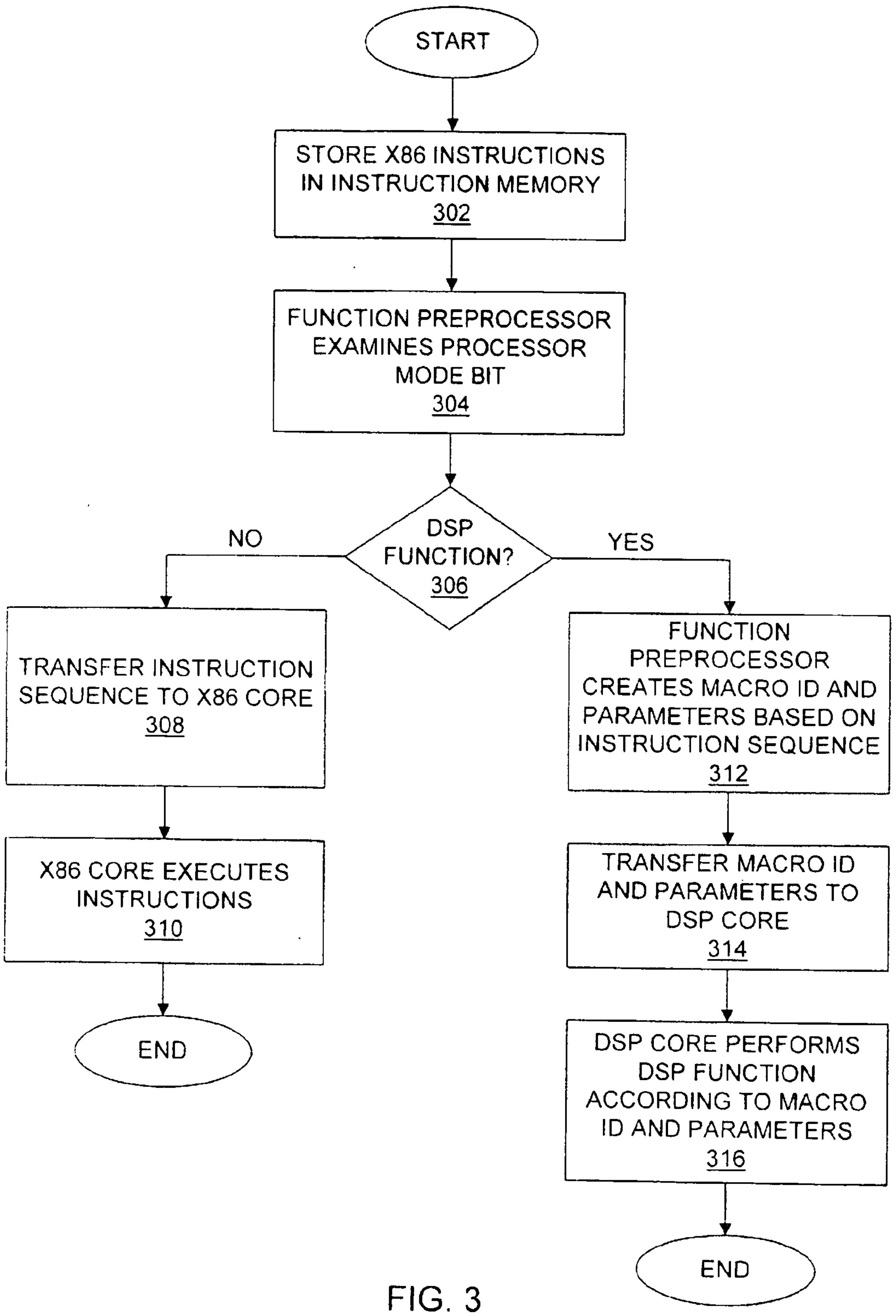

START
STORE X86 INSTRUCTIONS IN INSTRUCTION MEMORY 302
FUNCTION PREPROCESSOR EXAMINES PROCESSOR MODE BIT 304
DSP FUNCTION? 306
NO
YES
TRANSFER INSTRUCTION SEQUENCE TO X86 CORE 308
X86 CORE EXECUTES INSTRUCTIONS 310
END
FUNCTION PREPROCESSOR CREATES MACRO ID AND PARAMETERS BASED ON INSTRUCTION SEQUENCE 312
TRANSFER MACRO ID AND PARAMETERS TO DSP CORE 314
DSP CORE PERFORMS DSP FUNCTION ACCORDING TO MACRO ID AND PARAMETERS 316
END

FIG. 3

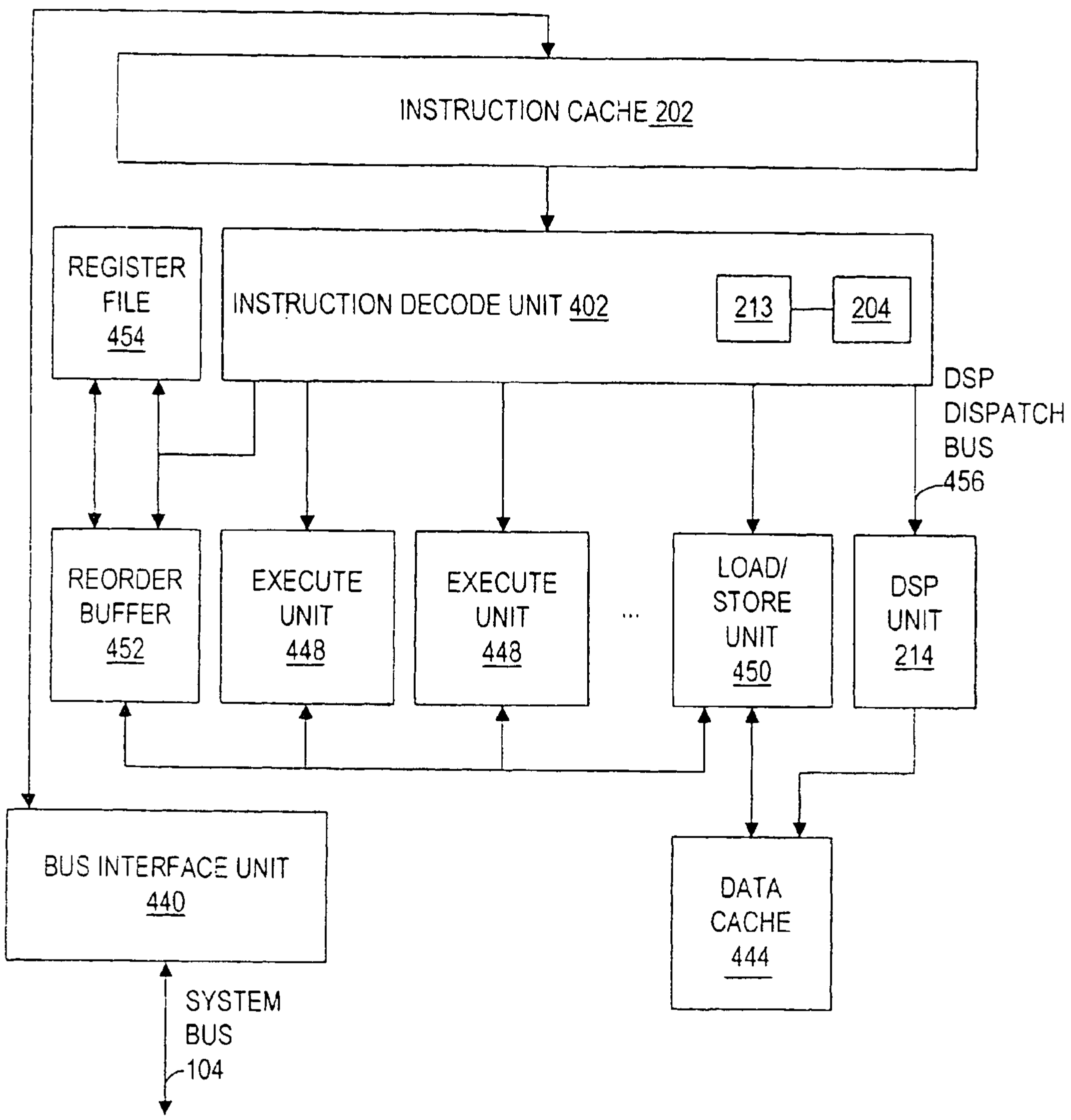

INSTRUCTION CACHE 202
REGISTER FILE 454
INSTRUCTION DECODE UNIT 402
213
204
DSP DISPATCH BUS 456
REORDER BUFFER 452
EXECUTE UNIT 448
EXECUTE UNIT 448
...
LOAD/ STORE UNIT 450
DSP UNIT 214
BUS INTERFACE UNIT 440
DATA CACHE 444
SYSTEM BUS 104

FIG. 4

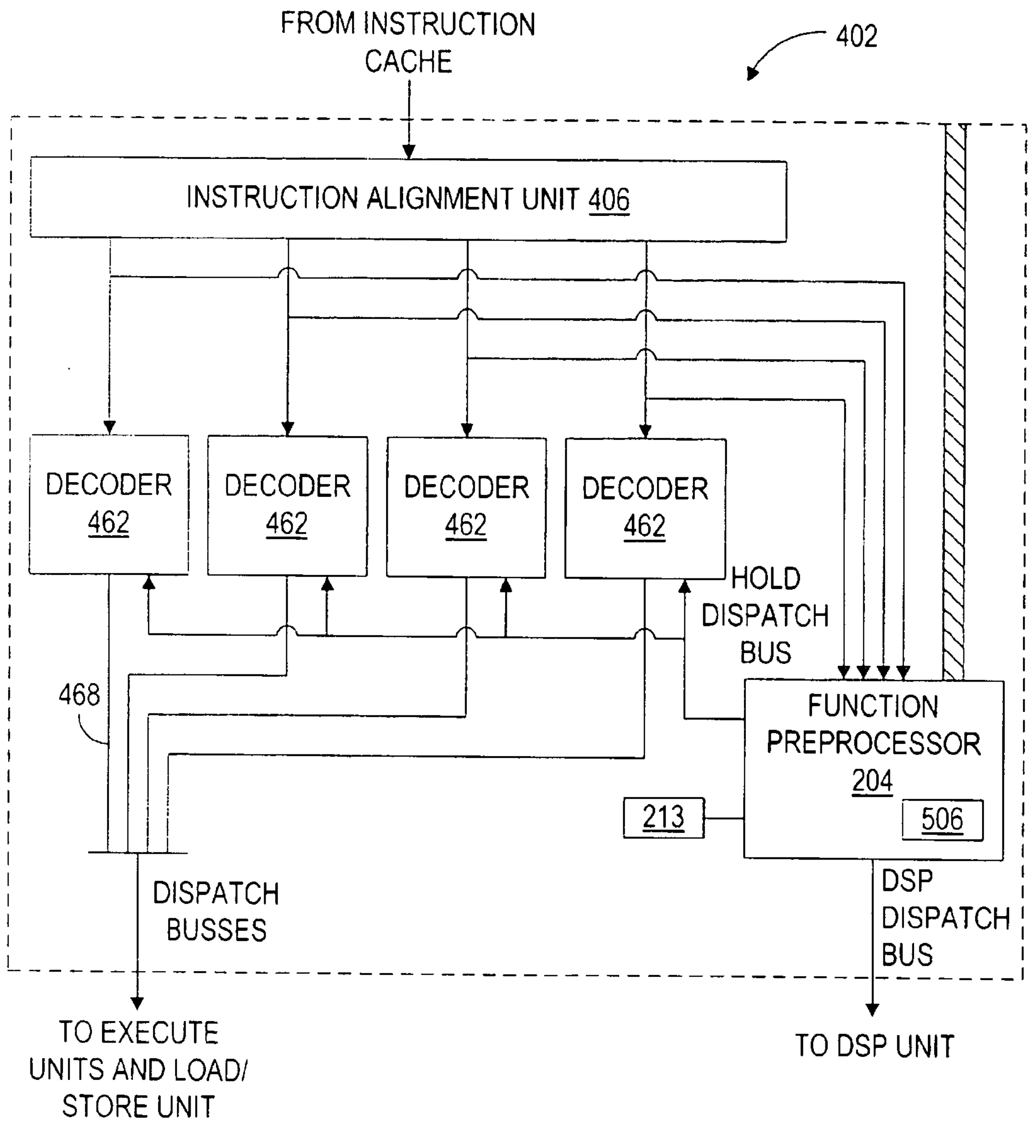

FROM INSTRUCTION CACHE
402
INSTRUCTION ALIGNMENT UNIT 406
DECODER 462
DECODER 462
DECODER 462
DECODER 462
HOLD DISPATCH BUS
468
FUNCTION PREPROCESSOR 204
506
213
DISPATCH BUSSES
DSP DISPATCH BUS
TO EXECUTE UNITS AND LOAD/ STORE UNIT
TO DSP UNIT

FIG. 5

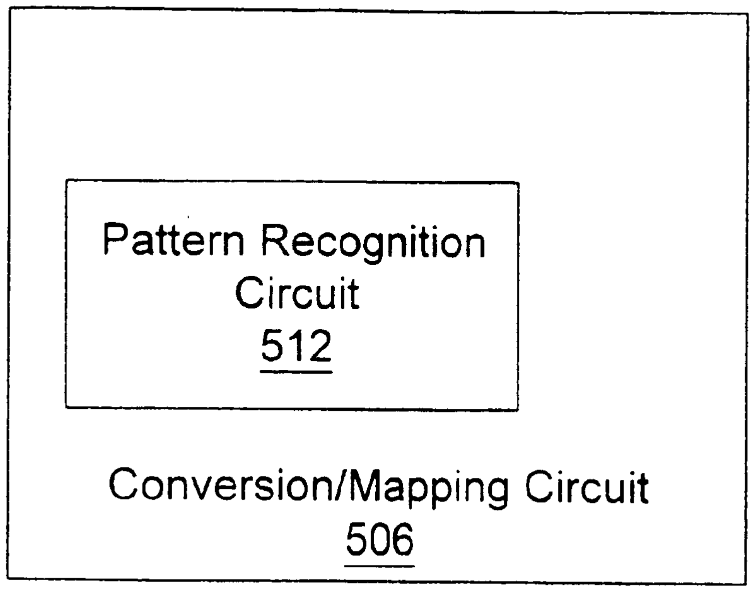

Pattern Recognition Circuit
512
Conversion/Mapping Circuit
506

FIG. 6

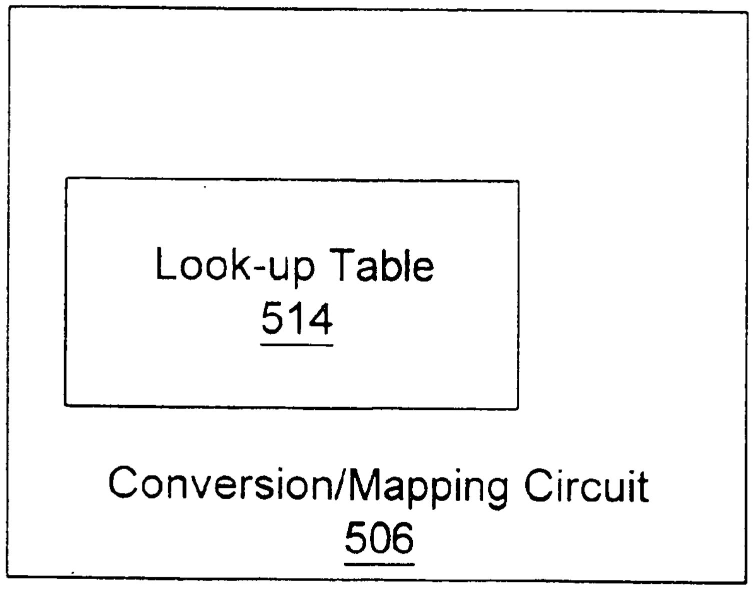

Look-up Table
514
Conversion/Mapping Circuit
506

FIG. 8

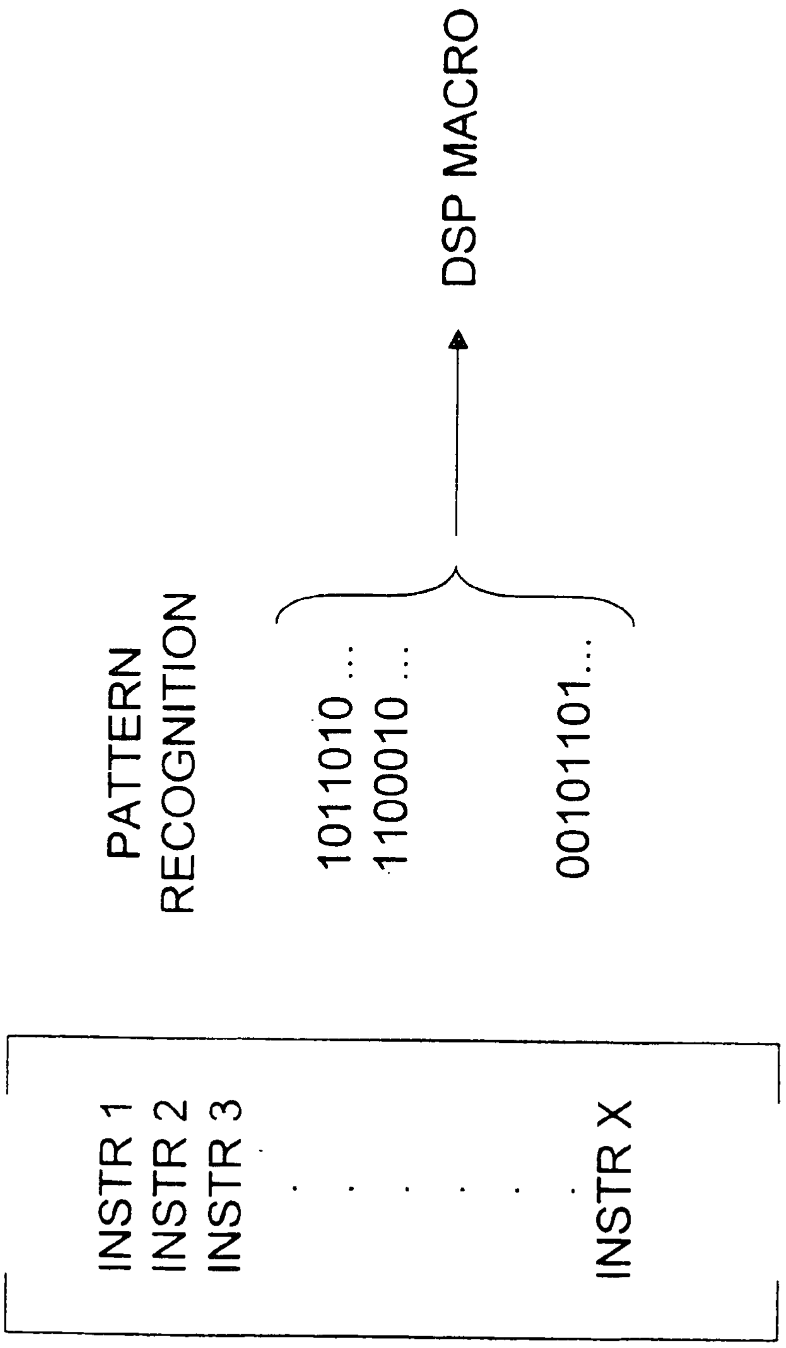

DSP MACRO
PATTERN RECOGNITION
1011010 ...
1100010 ...
00101101...
INSTR 1
INSTR 2
INSTR 3
INSTR X

FIG. 7

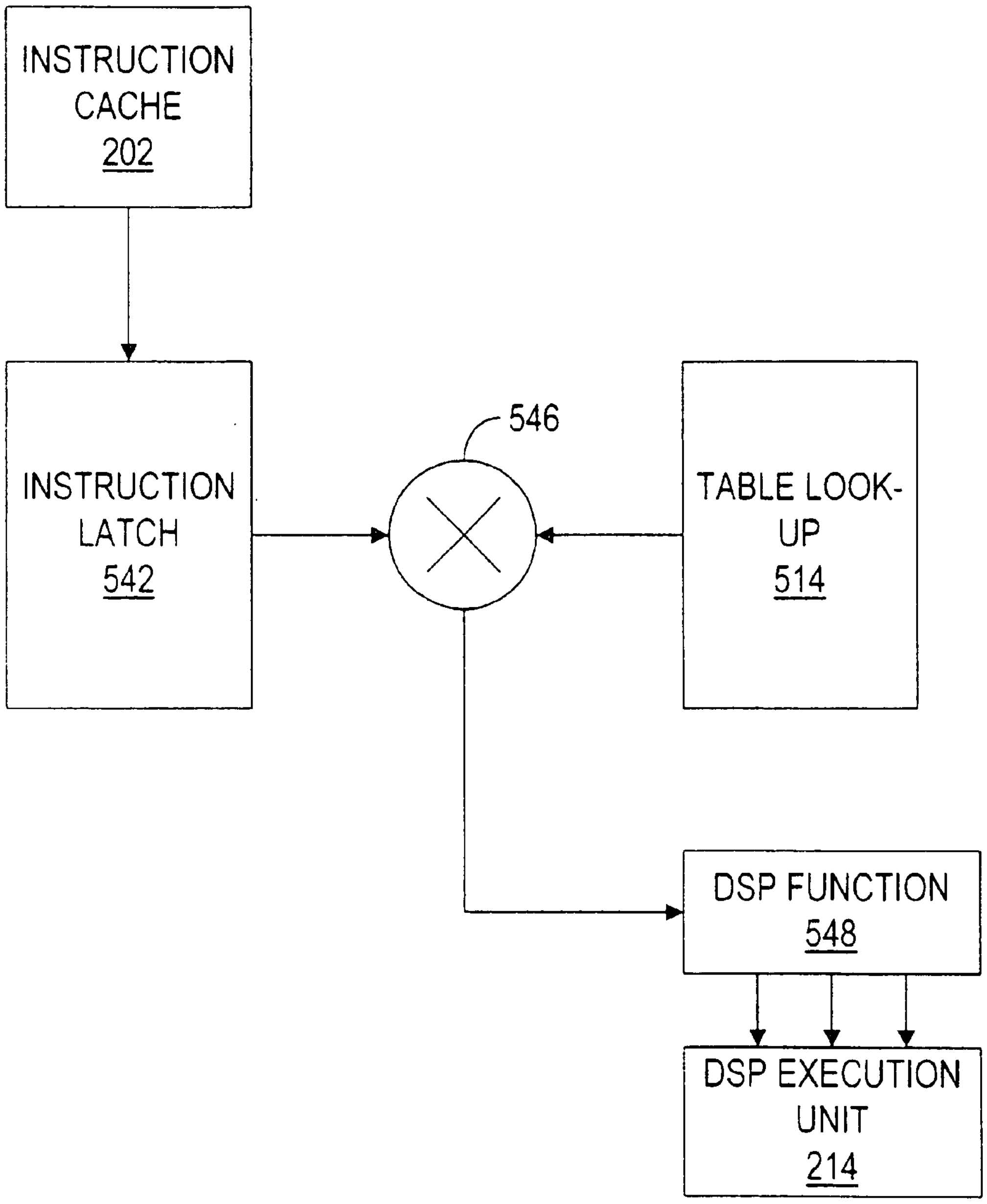

INSTRUCTION CACHE 202
INSTRUCTION LATCH 542
546
TABLE LOOK-UP 514
DSP FUNCTION 548
DSP EXECUTION UNIT 214

FIG. 9

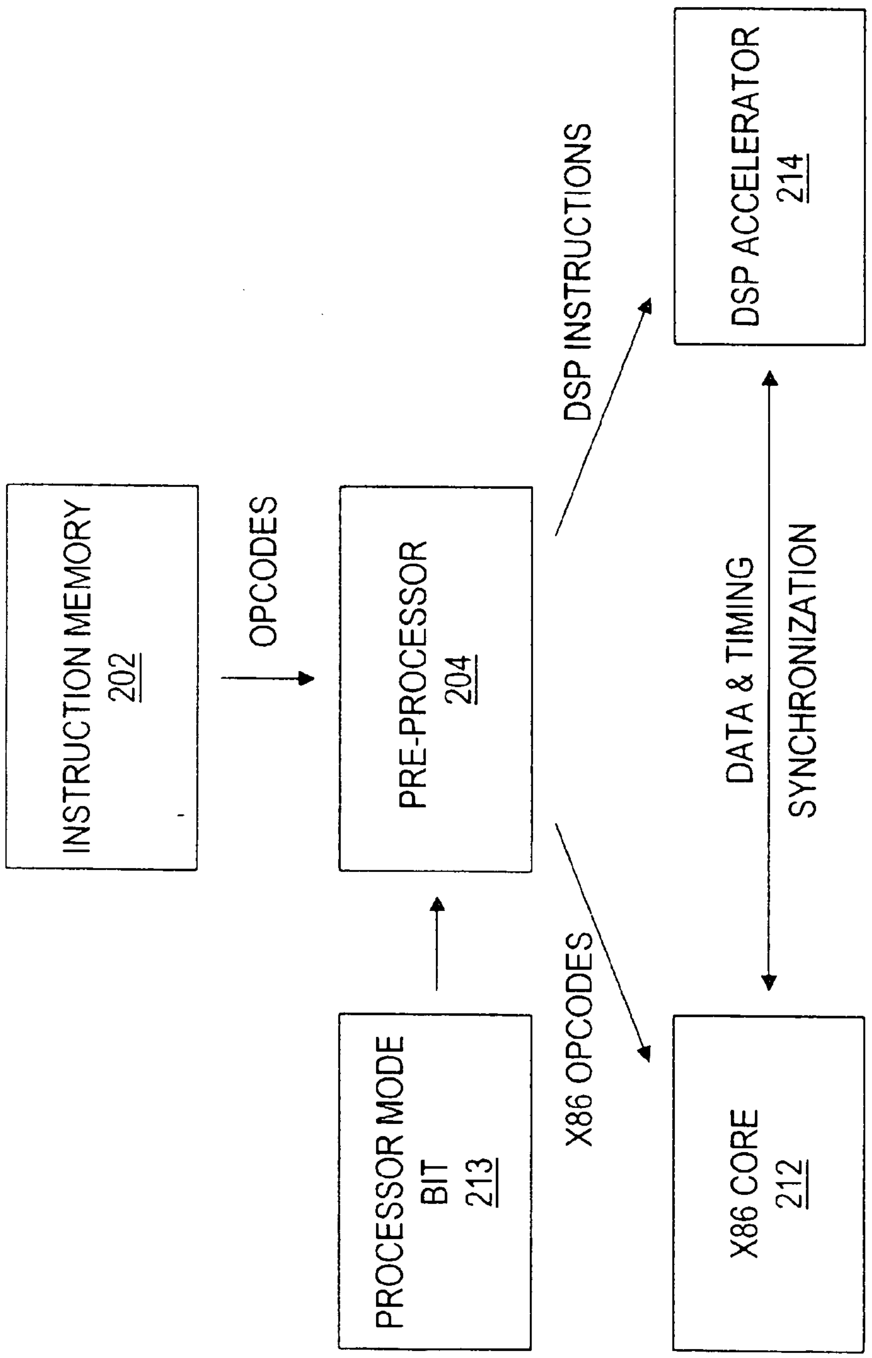

INSTRUCTION MEMORY 202
OPCODES
PRE-PROCESSOR 204
PROCESSOR MODE BIT 213
DSP INSTRUCTIONS
X86 OPCODES
DSP ACCELERATOR 214
DATA & TIMING SYNCHRONIZATION
X86 CORE 212

FIG. 10

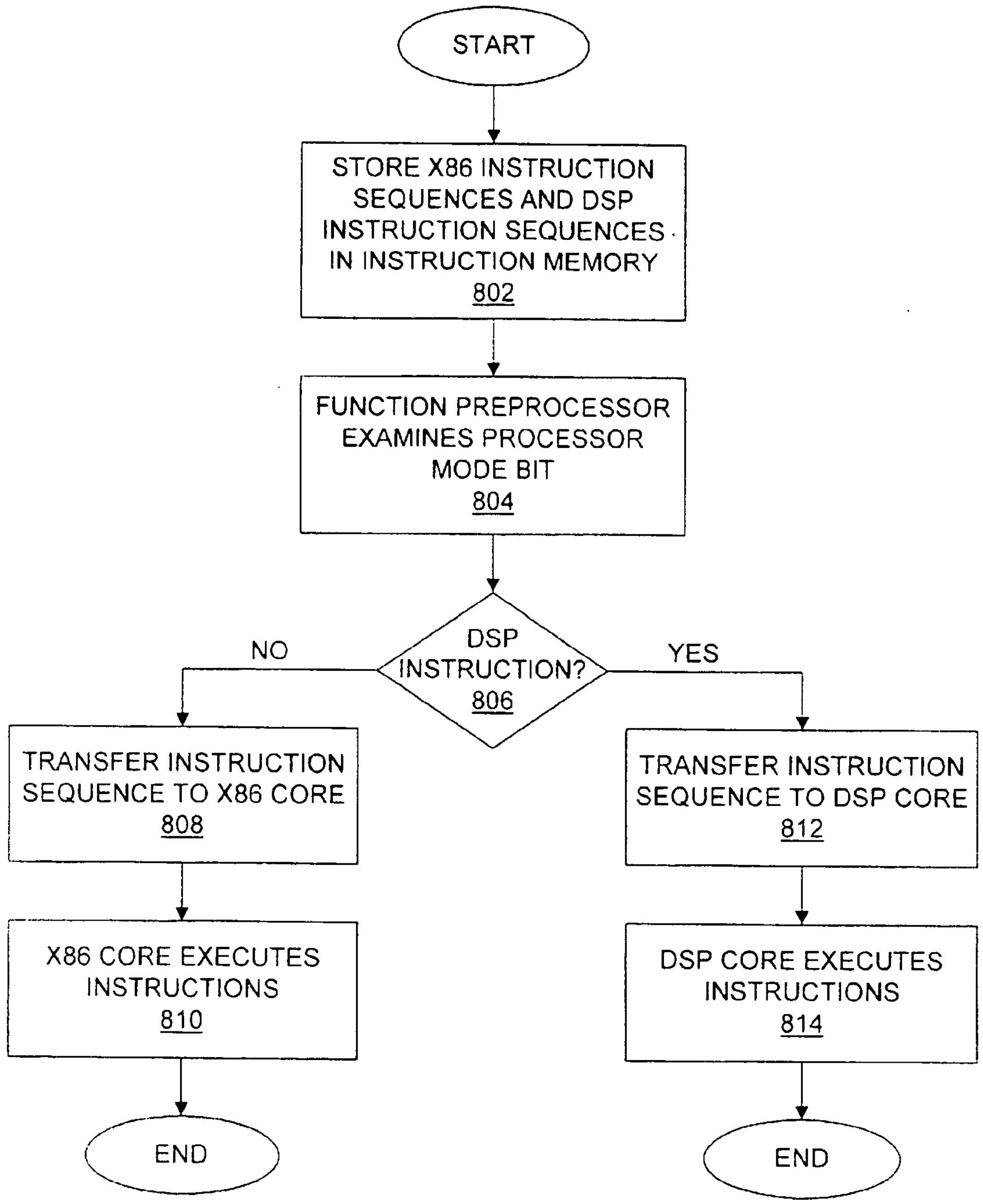

START
STORE X86 INSTRUCTION SEQUENCES AND DSP INSTRUCTION SEQUENCES IN INSTRUCTION MEMORY 802
FUNCTION PREPROCESSOR EXAMINES PROCESSOR MODE BIT 804
DSP INSTRUCTION? 806
NO
YES
TRANSFER INSTRUCTION SEQUENCE TO X86 CORE 808
X86 CORE EXECUTES INSTRUCTIONS 810
END
TRANSFER INSTRUCTION SEQUENCE TO DSP CORE 812
DSP CORE EXECUTES INSTRUCTIONS 814
END

FIG. 11

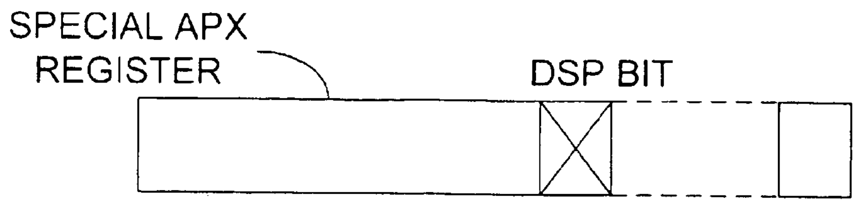

SPECIAL APX REGISTER
DSP BIT

FIG. 12

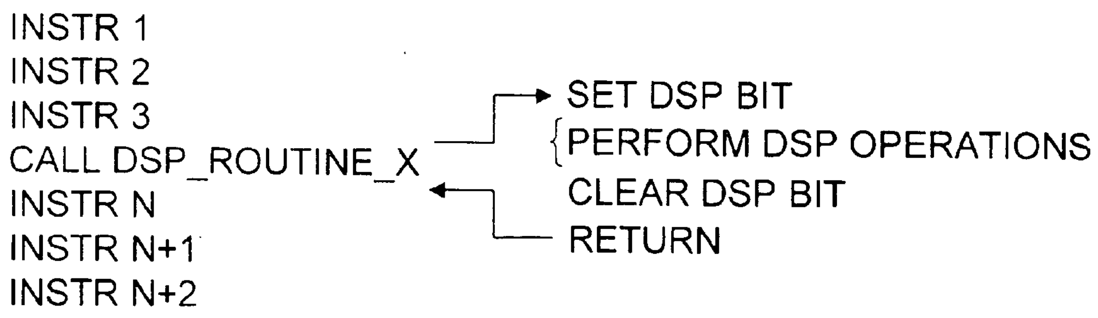

INSTR 1
INSTR 2
INSTR 3
CALL DSP_ROUTINE_X
INSTR N
INSTR N+1
INSTR N+2
SET DSP BIT
PERFORM DSP OPERATIONS
CLEAR DSP BIT
RETURN

FIG. 13

CENTRAL PROCESSING UNIT INCLUDING APX AND DSP CORES AND INCLUDING SELECTABLE APX AND DSP EXECUTION MODES

CONTINUATION DATA

This is a continuation-in-part of application Ser. No. 08/618,243 titled "Central Processing Unit Having an X86 and DSP Core and Including a DSP Function Decoder which Maps X86 Instructions to DSP Instructions" and filed Mar. 18, 1996, and which is assigned to Advanced Micro Devices Corp now U.S. Pat. No. 5,794,068.

CROSS REFERENCE TO RELATED APPLICATIONS

The following applications are related to the present application and are hereby incorporated by reference in their entirety.

U.S. patent application Ser. No. 08/618,243, titled "Central Processing Unit Having an X86 and DSP Core and Including a DSP Function Decoder which Maps X86 Instructions to DSP Instructions" and filed Mar. 18, 1996, now U.S. Pat. No. 5,794,068.

U.S. patent application Ser. No. 08/618,000, titled "Central Processing Unit Having X86 and DSP Functional Units" and filed Mar. 18, 1996, now U.S. Pat. No. 5,781,792.

U.S. patent application Ser. No. 08/618,242, titled "Central Processing Unit Including a DSP Function Preprocessor Having a Pattern Recognition Detector for Detecting Instruction Sequences which Perform DSP Functions" and filed Mar. 18, 1996, now U.S. Pat. No. 5,754,878.

U.S. patent application Ser. No. 08/618,241, titled "Central Processing Unit Including a DSP Function Preprocessor Having a Look-up Table Apparatus for Detecting Instruction Sequences which Perform DSP Functions" and filed Mar. 18, 1996, now U.S. Pat. No. 5,784,640.

U.S. patent application Ser. No. 08/618,240, titled "Central Processing Unit Including a DSP Function Preprocessor Which Scans Instruction Sequences for DSP Functions" and filed Mar. 18, 1996, now U.S. Pat. No. 5,790,824.

The above related applications are all assigned to Advanced Micro Devices, Inc.

FIELD OF THE INVENTION

The present invention relates to a computer system CPU or microprocessor which includes a general purpose core and a DSP core, wherein the CPU includes a switch for selecting a processor execution mode to selectively enable processing of DSP instructions.

DESCRIPTION OF THE RELATED ART

Personal computer systems and general purpose microprocessors were originally developed for business applications such as word processing and spreadsheets, among others. However, computer systems are currently being used to handle a number of real time DSP-related applications, including multimedia applications having video and audio components, video capture and playback, telephony applications, speech recognition and synthesis, and communication applications, among others. These real time or DSP-like applications typically require increased CPU floating point performance.

One problem that has arisen is that general purpose microprocessors originally designed for business applications are not well suited for the real-time requirements and mathematical computation requirements of modem DSP-related applications, such as multimedia applications and communications applications. For example, the X86 family of microprocessors from Intel Corporation are oriented toward integer-based calculations and memory management operations and do not perform DSP-type functions very well.

As personal computer systems have evolved toward more real-time and multimedia capable systems, the general purpose CPU has been correspondingly required to perform more mathematically intensive DSP-type functions. Therefore, many computer systems now include one or more digital signal processors which are dedicated towards these complex mathematical functions.

A recent trend in computer system architectures is the movement toward "native signal processing (NSP)". Native signal processing or NSP was originally introduced by Intel Corporation as a strategy to offload certain functions from DSPs and perform these functions within the main or general purpose CPU. The strategy presumes that, as performance and clock speeds of general purpose CPUs increase, the general purpose CPU is able to perform many of the functions formerly performed by dedicated DSPs. Thus, one trend in the microprocessor industry is an effort to provide CPU designs with higher speeds and augmented with DSP-type capabilities, such as more powerful floating point units. Another trend in the industry is for DSP manufacturers to provide DSPs that not only run at high speeds but also can emulate CPU-type capabilities such as memory management functions.

A digital signal processor is essentially a general purpose microprocessor which includes special hardware for executing mathematical functions at speeds and efficiencies not usually associated with microprocessors. In current computer system architectures, DSPs are used as co-processors and operate in conjunction with general purpose CPUs within the system. For example, current computer systems may include a general purpose CPU as the main CPU and include one or more multimedia or communication expansion cards which include dedicated DSPs. The CPU offloads mathematical functions to the digital signal processor, thus increasing system efficiency.

Digital signal processors include execution units that comprise one or more arithmetic logic units (ALUs) coupled to hardware multipliers which implement complex mathematical algorithms in a pipelined manner. The instruction set primarily comprises DSP-type instructions and also includes a small number of instructions having non-DSP functionality.

The DSP is typically optimized for mathematical algorithms such as correlation, convolution, finite impulse response (FIR) filters, infinite impulse response (IIR) filters, Fast Fourier Transforms (FFTs), matrix computations, and inner products, among other operations. Implementations of these mathematical algorithms generally comprise long sequences of systematic arithmetic/multiplicative operations. These operations are interrupted on various occasions by decision-type commands. In general, the DSP sequences are a repetition of a very small set of instructions that are executed 70% to 90% of the time. The remaining 10% to 30% of the instructions are primarily Boolean/decision operations (or general data processing).

A general purpose CPU is comprised of an execution unit, a memory management unit, and a floating point unit, as well as other logic. The task of a general purpose CPU is to execute code and perform operations on data in the computer memory and thus to manage the computing platform. In general, the general purpose CPU architecture is designed primarily to perform Boolean/management/data manipulation decision operations. The instructions or opcodes executed by a general-purpose CPU include basic mathematical functions. However these mathematical functions are not well adapted to complex DSP-type mathematical operations. Thus a general purpose CPU is required to execute a large number of opcodes or instructions to perform basic DSP functions.

Therefore, a computer system and CPU architecture is desired which includes a general purpose CPU and which also performs DSP-type mathematical functions with increased performance. A CPU architecture is also desired which is backwards compatible with existing software applications which presume that the general purpose CPU is performing all of the mathematical computations. A new CPU architecture is further desired which provides increased mathematical performance for existing software applications.

One popular microprocessor used in personal computer systems is the X86 family of microprocessors. The X86 family of microprocessors includes the 8088, 8086, 80186, 80286, 80386, i486, Pentium, and P6 microprocessors from Intel Corporation. The X86 family of microprocessors also includes X86 compatible processors such as the 4486 and K5 processors from Advanced Micro Devices, the M1 processor from Cyrix Corporation, and the NextGen 5x86 and 6x86 processors from NextGen Corporation. The X86 family of microprocessors was primarily designed and developed for business applications. In general, the instruction set of the X86 family of microprocessors does not include sufficient mathematical or DSP functionality for modem multimedia and communications applications. Therefore, a new X86 CPU architecture is further desired which implements DSP functions more efficiently than current X86 processors. It would further be desirable that this new X86 CPU architecture did not require additional opcodes for the X86 processor.

SUMMARY OF THE INVENTION

The present invention comprises a CPU or microprocessor which includes a general purpose CPU component, such as an X86 core, and also includes a DSP core. The CPU includes a switch for selecting a processor execution mode. The switch selectively enables processing of general purpose instructions, e.g., APX instructions, or DSP instructions. In the preferred embodiment comprising an APX-based CPU, the CPU includes one or more bits, referred to as processor mode bits, that are set to indicate whether the instruction decode engine should interpret the incoming code sequence as DSP instructions or APX instructions. Thus, for example, the processor mode bit is set to indicate a sequence of DSP instructions, and the processor mode bit is cleared to indicate that the program sequence reverts back to a normal APX mode of operation. The CPU may include other means for indicating or differentiating between APX and DSP instructions, as desired. The CPU includes a preprocessor which examines the processor mode bit and selectively provides instructions to either the X86 core or the DSP.

In a first embodiment, the CPU receives only APX instructions. In this first embodiment, the CPU includes an intelligent DSP function decoder or preprocessor which examines sequences of APX instructions or opcodes (X86

opcodes) and converts or maps the instruction sequence to a DSP macro instruction or function identifier that is provided to the DSP core. The processor mode bit is set to indicate the start of an APX code sequence which implements a DSP function. The preprocessor thus examines the processor mode bit to determine if a DSP function is being executed. If the preprocessor determines that a DSP function is being executed based on the processor mode bit, the preprocessor converts or maps the instruction sequence to a DSP macro instruction or function identifier that is provided to the DSP core. The DSP core executes one or more DSP instructions to implement the desired DSP function indicated by the DSP macro or function identifier. The DSP core preferably performs the DSP function in parallel with other operations performed by the general purpose CPU core for increased system performance.

In one embodiment, the CPU includes a processor mode register which stores the processor mode bit, and also includes one or more bits, preferably a plurality of bits, which identify the type of DSP function implemented by the instruction sequence. Thus, the preprocessor examines the processor mode bit to determine if the APX code sequence implements a DSP function. If so, the preprocessor examines the plurality of bits to determine the general type of DSP function being implemented. The preprocessor uses the information on the general type of DSP function in creating the function identifier, and the preprocessor also examines the instruction sequence to extract values and parameters necessary for the DSP core to implement the DSP function.

In a second embodiment, the CPU receives an instruction sequence which comprises sequences of general purpose, e.g., APX instructions, and which also comprises sequences of DSP instructions. The respective processor mode bit is set to indicate the beginning of a sequence of DSP instructions, and the processor mode bit is cleared to indicate the beginning of a sequence of APX instructions. The CPU thus routes the instructions to the APX core or the DSP core based on the status of the processor mode bit.

BRIEF DESCRIPTION OF THE DRAWINGS

A better understanding of the present invention can be obtained when the following detailed description of the preferred embodiment is considered in conjunction with the following drawings, in which:

FIG. 1 is a block diagram of a computer system including a CPU having a general purpose CPU core and a DSP core according to the present invention

FIG. 2 is a block diagram of the CPU of FIG. 1 including a general purpose CPU core and a DSP core and including a DSP function preprocessor according to the present invention;

FIG. 3 is a flowchart diagram illustrating operation of the present invention;

FIG. 4 is a more detailed block diagram of the CPU of FIG. 1;

FIG. 5 is a block diagram of the Instruction Decode Unit of FIG. 4;

FIG. 6 is a block diagram of the function preprocessor including a pattern recognition detector according to one embodiment of the invention;

FIG. 7 illustrates operation of the pattern recognition detector of FIG. 6;

FIG. 8 is a block diagram of the function preprocessor including a look-up table according to one embodiment of the invention;

FIG. 9 illustrates operation of the look-up table of FIG. 8; and

FIG. 10 is a block diagram diagram of the CPU according to the second emobodiment.

FIG. 11 is a flowchart diagram illustrating a second embodiment of the present invention.

FIG. 12 illustrates one embodiment of the processor mode register.

FIG. 13 illustrates one embodiment of an instruction sequence which includes a DSP instruction sequence.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

Incorporation by Reference

Pentium System Architecture by Don Anderson and Tom Shanley and available from Mindshare Press, 2202 Buttercup Dr., Richardson, Tex. 75082 (214) 231–2216, is hereby incorporated by reference in its entirety.

Digital Signal Processing Applications Using the ADSP-2100 Family Volumes 1 and 2, 1995 edition, available from Analog Devices Corporation of Norwood Mass., is hereby incorporated by reference in its entirety.

The Intel CPU Handbook, 1994 and 1995 editions, available from Intel Corporation, are hereby incorporated by reference in their entirety.

The AMD K5 Handbook, 1995 edition, available from Advanced Micro Devices Corporation, is hereby incorporated by reference in its entirety.

Computer System Block Diagram

Referring now to FIG. 1, a block diagram of a computer system incorporating a central processing unit (CPU) or microprocessor 102 according to the present invention is shown. The computer system shown in FIG. 1 is illustrative only, and the CPU 102 of the present invention may be incorporated into any of various types of computer systems.

As shown, the CPU 102 includes a general purpose CPU core 212 and a DSP core 214. The general purpose core 212 executes general purpose (non-DSP) opcodes and the DSP core 214 executes DSP-type functions, as described further below. In the preferred embodiment, the general purpose CPU core 212 is an X86 core, i.e., is compatible with the X86 family of microprocessors. However, the general purpose CPU core 212 may be any of various types of CPUs, including the PowerPC family, the DEC Alpha, and the SunSparc family of processors, among others. In the following disclosure, the general purpose CPU core 212 is referred to as an X86 core for convenience. The general purpose core 212 may comprise one or more general purpose execution units, and the DSP core 214 may comprise one or more digital signal processing execution units.

As discussed further below, the CPU includes a switch 213 for selecting a processor execution mode. The switch 213 selectively enables processing of general purpose instructions, e.g., APX instructions, or DSP instructions. In the preferred embodiment comprising an APX-based CPU, the CPU includes one or more bits in a register, referred to as processor mode bits, that are set to indicate whether the instruction decode engine should interpret the incoming code sequence as DSP instructions or APX instructions. Thus, for example, the processor mode bit is set to indicate a sequence of DSP instructions, and the processor mode bit is cleared to indicate that the program sequence reverts back to a normal APX mode of operation. The CPU 102 may include other means for indicating or differentiating between APX and DSP instructions, as desired.

The CPU 102 also includes a preprocessor 204 which examines the processor mode bit and selectively provides instructions to either the X86 core 212 or the DSP 214.

As shown, the CPU 102 is coupled through a CPU local bus 104 to a host/PCI/cache bridge or chipset 106. The chipset 106 is preferably similar to the Triton chipset available from Intel Corporation. A second level or L2 cache memory (not shown) may be coupled to a cache controller in the chipset, as desired. Also, for some processors the external cache may be an L1 or first level cache. The bridge or chipset 106 couples through a memory bus 108 to main memory 110. The main memory 110 is preferably DRAM (dynamic random access memory) or EDO (extended data out) memory, or other types of memory, as desired.

The chipset 106 includes various peripherals, including an interrupt system, a real time clock (RTC) and timers, a direct memory access (DMA) system, ROM/Flash memory, communications ports, diagnostics ports, command/status registers, and non-volatile static random access memory (NVSRAM) (all not shown).

The host/PCI/cache bridge or chipset 106 interfaces to a peripheral component interconnect (PCI) bus 120. In the preferred embodiment, a PCI local bus is used. However, it is noted that other local buses may be used, such as the VESA (Video Electronics Standards Association) VL bus. Various types of devices may be connected to the PCI bus 120. In the embodiment shown in FIG. 1, a video/graphics controller or adapter 170 and a network interface controller 140 are coupled to the PCI bus 120. The video adapter connects to a video monitor 172, and the network interface controller 140 couples to a local area network (LAN). A SCSI (small computer systems interface) adapter 122 may also be coupled to the PCI bus 120, as shown. The SCSI adapter 122 may couple to various SCSI devices 124, such as a CD-ROM drive and a tape drive, as desired. Various other devices may be connected to the PCI bus 120, as is well known in the art.

Expansion bus bridge logic 150 may also be coupled to the PCI bus 120. The expansion bus bridge logic 150 interfaces to an expansion bus 152. The expansion bus 152 may be any of varying types, including the industry standard architecture (ISA) bus, also referred to as the AT bus, the extended industry standard architecture (EISA) bus, or the MicroChannel architecture (MCA) bus. Various devices may be coupled to the expansion bus 152, such as expansion bus memory 154 and a modem 156.

CPU Block Diagram

Referring now to FIG. 2, a high level block diagram illustrating certain components in the CPU 102 of FIG. 1 is shown. As shown, the CPU 102 includes an instruction cache or instruction memory 202 which receives instructions or opcodes from the system memory 110. Function preprocessor 204 is coupled to the instruction memory 202 and examines instruction sequences or opcode sequences in the instruction memory 202. The function preprocessor 204 is also coupled to the X86 core 212 and the DSP core 214. The function preprocessor 204 is further coupled to the processor mode register 213 storing the processor mode bit. As shown, the function preprocessor 204 examines the processor mode bit and selectively provides instructions or opcodes to either the X86 core 212 or selectively provides op-codes or information to the DSP core 214.

The X86 core 212 and DSP core 214 are coupled together and provide data and timing signals between each other. In one embodiment, the CPU 102 includes one or more buffers (not shown) which interface between the X86 core 212 and the DSP core 214 to facilitate transmission of data between the X86 core 212 and the DSP core 214.

In a first embodiment, the CPU 102 receives only APX instructions. In this first embodiment, if the processor mode bit is set to indicate DSP functions, the function preprocessor **204** examines the sequences of APX instructions or opcodes (X86 opcodes) and converts or maps the instruction sequence to a DSP macro instruction or function identifier that is provided to the DSP core. The processor mode bit is thus set to indicate the start of an APX code sequence which implements a DSP function. The function preprocessor **204** examines the processor mode bit to determine if a DSP function is being executed by the APX code sequence. If the function preprocessor **204** determines that a DSP function is being executed based on the processor mode bit, the function preprocessor **204** converts or maps the instruction sequence to a DSP macro instruction or function identifier that is provided to the DSP core **214**. The DSP core **214** executes one or more DSP instructions to implement the desired DSP function indicated by the DSP macro or function identifier. The DSP core **214** preferably performs the DSP function in parallel with other operations performed by the general purpose CPU core **212** for increased system performance.

In one embodiment, the processor mode register **213** stores the processor mode bit, and also includes one or more bits, preferably a plurality of bits, which identify the type of DSP function implemented by the instruction sequence. Thus, the preprocessor **204** examines the processor mode bit to determine if the APX code sequence implements a DSP function. If so, the preprocessor **204** examines the plurality of bits to determine the general type of DSP function being implemented. The preprocessor **204** uses the information on the general type of DSP function in creating the function identifier, and the preprocessor **204** also examines the instruction sequence to extract values and parameters necessary for the DSP core to implement the DSP function.

In a second embodiment, the CPU **212** receives an instruction sequence which comprises sequences of general purpose, e.g., APX instructions, and which also comprises sequences of DSP instructions. The respective processor mode bit is set to indicate the beginning of a sequence of DSP instructions, and the processor mode bit is cleared to indicate the beginning of a sequence of APX instructions. The pre-processor **204** thus routes the instructions to the APX core or the DSP core based on the status of the processor mode bit. In this embodiment, the pre-processor **204** is not required to map APX instructions into DSP macros, but rather simply routes APX instructions to the x 86 core **212** and routes DSP instructions to the DSP core **214** based on the status of the processor mode bit.

FIG. 3—Flowchart

Referring now to FIG. **3**, a flowchart diagram illustrating operation of the first embodiment of the present invention is shown. It is noted that two or more of the steps in FIG. **3** may operate concurrently, and the operation of the invention is shown in flowchart form for convenience.

As shown, in step **302** the instruction memory **202** receives and stores a plurality of X86 instructions. The plurality of X86 instructions may include one or more instruction sequences which implement a DSP function.

In step **304** the function preprocessor **204** analyzes the processor mode bit. The value of the processor mode bit is preferably set by the program, i.e., the program which comprises the instruction sequences being examined. As noted above, in the first embodiment, the processor mode bit is set to indicate that the sequence of instructions are designed or intended to perform a DSP-type function. The processor mode bit is cleared to indicate that the sequence of instructions are a regular sequence of X86 instructions that are not intended to perform a DSP-type function. In the present disclosure, a DSP-type function comprises one or more of the following mathematical functions: correlation, convolution, Fast Fourier Transform, finite impulse response filter, infinite impulse response filter, inner product, and matrix manipulation, among others.

In step **306** the function preprocessor **204** determines, based on the status of the processor mode bit, if the sequence of instructions are designed or intended to perform a DSP-type function.

If the processor mode bit is cleared to indicate that the instructions or opcodes stored in the instruction cache **202** do not correspond to a DSP-type function, the instructions are provided to the X86 core **212** in step **308**. Thus, these instructions or opcodes are provided directly from the instruction cache **202** to the X86 core **212** for execution, as occurs in prior art X86 compatible CPUs. After the opcodes are transferred to the X86 core **212**, in step **310** the X86 core **212** executes the instructions.

If the processor mode bit is set to indicate that the sequence of instructions correspond to or implement a DSP-type function in step **306**, then in step **312** the function preprocessor **204** analyzes the sequence of instructions and determines the respective DSP-type function being implemented. In step **312** the function preprocessor **204** maps the sequence of instructions to a respective DSP macro identifier, also referred to as a function identifier. The function preprocessor **204** also analyzes the information in the sequence of opcodes in step **312** and generates zero or more parameters for use by the DSP core or accelerator **214** in executing the function identifier.

As described above, in one embodiment of the invention, the processor mode register **213** stores a processor mode bit and in addition stores one or more bits, preferably a plurality of bits, which indicate the general type of DSP function being performed. Thus the application program writes a value into the processor mode register indicating the type of DSP function being implemented by the APX instruction sequence. In this embodiment, in step **312** the preprocessor **204** uses the value indicating the type of DSP function to aid in converting the sequence of instructions into a DSP function identifier and zero or more parameters. Thus, in this embodiment, the preprocessor **204** examines the processor mode bit in step **304** to determine if the APX code sequence implements a DSP function. If so, in step **312** the preprocessor **204** examines the plurality of bits to determine the general type of DSP function being implemented. The preprocessor **204** then examines the instruction sequence in step **312** to extract values and parameters necessary for the DSP core to implement the DSP function.

As shown, after the preprocessor **204** has generated the function identifier and the parameters in step **312**, in step **314** the function preprocessor **204** provides the function identifier and the parameters to the DSP core **214**.

The DSP core **214** receives the function identifier and the associated parameters from the function preprocessor **204** and in step **316** performs the respective DSP function. In the preferred embodiment, the DSP core **214** uses the function identifier to index into a DSP microcode RAM or ROM to execute a sequence of DSP instructions or opcodes. The DSP instructions cause the DSP to perform the desired DSP-type function. The DSP core **214** also uses the respective parameters in executing the DSP function.

As mentioned above, the X86 core **212** and DSP core **214** are coupled together and provide data and timing signals between each other. In the preferred embodiment, the X86 core **212** and DSP core **214** operate substantially in parallel. Thus, while the X86 core **212** is executing one sequence of opcodes, the DSP accelerator **214** may be executing one or more DSP functions corresponding to another sequence of opcodes. Thus, the DSP core **214** does not operate as a slave or co-processor, but rather operates as an independent execution unit or pipeline. The DSP core **214** and the X86 core **212** provide data and timing signals to each other to indicate the status of operations and also to provide any data outputs produced, as well as to ensure data coherency/independence.

Example Operation

The following describes an example of how a string or sequence of X86 opcodes are converted into a function identifier and then executed by the DSP core or accelerator **214** according to the present invention. The following describes an X86 opcode sequence which performs a simple inner product computation, wherein the inner product is averaged over a vector comprising 20 values:

X86 Code
(Simple inner product)

1	Mov ECX, num__samples:	{Set up parameters for macro}
1	Mov ESI, address__1	
1	Mov EDI, address__2	
1	Mov EAX, 0:	{Initialize vector indices}
1	Mov EBX, 0:	
4	FLdZ:	{Initialize sum of products}
	Again:	
		{Update counter}
4	Fld dword ptr [ESI+EAX*4]:	{Get vector elements and}
1	Inc EAX:	{update indices}
4	Fld dword ptr [EDI+EBX*4]:	
1	Inc EBX:	
13	FMulP St(1), St:	{Compute product term}
7	FAddP St(1), St:	{Add term to sum}
1	LOOP Again:	{Continue if more terms}

As shown, the X86 opcode instructions for a simple inner product comprise a plurality of move instructions followed by an F-load function wherein this sequence is repeated a plurality of times. If this X86 opcode sequence were executed by the X86 core 212, the execution time for this inner product computation would require 709 cycles (9+20×35). This assumes i486 timing, concurrent execution of floating point operations, and cache hits for all instructions and data required for the inner product computation. The function preprocessor **204** analyzes the sequence of opcodes and detects that the opcodes are performing an inner product computation. The function preprocessor **204** then converts this entire sequence of X86 opcodes into a single macro or function identifier and one or more parameters. An example macro or function identifier that is created based on the X86 opcode sequence shown above would be as follows:

Example Macro
(as it appears in assembler)

Inner__product__simple (	
address__1,	{Data vector}
address__2,	{Data vector}
num__samples):	{Length of vector}

This function identifier and one or more parameters are provided to the DSP core **214**. The DSP core **214** uses the macro provided from the function preprocessor **204** to load one or more DSP opcodes or instructions which execute the DSP function. In the preferred embodiment, the DSP core **214** uses the macro to index into a ROM which contains the instructions used for executing the DSP function. In this

example, the DSP code or instructions executed by the DSP core **214** in response to receiving the macro described above are shown below:

DSP Code
(Simple inner product)

1	Cntr =num__samples:	{Set up parameters from macro}
1	ptrl =address__1:	
1	ptr2 =address__2:	
1	MAC =0:	{Initialize sum of products}
1	regl =*ptrl++, reg2 =*ptr2++:	{Pre-load multiplier input registers}
1	Do LOOP until ce:	{Specify loop parameters}
1	MAC +=regl *reg2, reg1 =*ptrl++, reg2 =*ptr2++;	{Form sum of products}
	LOOP:	{Continue if more terms}

In this example, the DSP core **214** performs this inner product averaged over a vector comprising 20 values and consumes a total of 26 cycles (6+20×1). This assumes typical DSP timing, including a single cycle operation of instructions, zero overhead looping and cache hits for all instructions and data. Thus, the DSP core **214** provides a performance increase of over 28 times of that where the X86 core **212** executes this DSP function.

FIG. **4**—CPU Block Diagram

Referring now to FIG. **4**, a more detailed block diagram is shown illustrating the internal components of the CPU **102** according to the present invention. Elements in the CPU **102** that are not necessary for an understanding of the present invention are not described for simplicity. As shown, in the preferred embodiment the CPU **102** includes a bus interface unit **440**, instruction cache **202**, a data cache **444**, an instruction decode unit **402**, a plurality of execute units **448**, a load/store unit **450**, a reorder buffer **452**, a register file **454**, and a DSP unit **214**.

As shown, the CPU **102** includes a bus interface unit **440** which includes circuitry for performing communication upon CPU bus **104**. The bus interface unit **440** interfaces to the data cache **444** and the instruction cache **202**. The instruction cache **202** prefetches instructions from the system memory **110** and stores the instructions for use by the CPU **102**. The instruction decode unit **402** is coupled to the instruction cache **202** and receives instructions from the instruction cache **202**. The instruction decode unit **402** includes function preprocessor **204** and processor mode register or bit **213**, as shown. The function preprocessor **204** in the instruction decode unit **402** is coupled to the instruction cache **202**. The instruction decode unit **402** further includes an instruction alignment unit as well as other logic.

The instruction decode unit **402** couples to a plurality of execution units **448**, reorder buffer **452**, and load/store unit **450**. The plurality of execute units are collectively referred to herein as execute units **448**. Reorder buffer **452**, execute units **448**, and load/store unit **450** are each coupled to a forwarding bus **458** for forwarding of execution results. Load/store unit **450** is coupled to data cache **444**. DSP unit **214** is coupled directly to the instruction decode unit **402** through the DSP dispatch bus **456**. It is noted that one or more DSP units **214** may be coupled to the instruction decode unit **402**.

Bus interface unit **440** is configured to effect communication between microprocessor **102** and devices coupled to system bus **104**. For example, instruction fetches which miss instruction cache **202** are transferred from main memory **110** by bus interface unit **440**. Similarly, data requests performed by load/store unit **450** which miss data cache **444** are transferred from main memory **110** by bus interface unit **440**. Additionally, data cache **444** may discard a cache line of data which has been modified by microprocessor **102**. Bus interface unit **440** transfers the modified line to main memory **110**.

Instruction cache **202** is preferably a high speed cache memory for storing instructions. It is noted that instruction cache **202** may be configured into a set-associative or direct mapped configuration. Instruction cache **202** may additionally include a branch prediction mechanism for predicting branch instructions as either taken or not taken. A "taken" branch instruction causes instruction fetch and execution to continue at the target address of the branch instruction. A "not taken" branch instruction causes instruction fetch and execution to continue at the instruction subsequent to the branch instruction. Instructions are fetched from instruction cache **202** and conveyed to instruction decode unit **402** for decode and dispatch to an execution unit. The instruction cache **202** may also include a macro prediction mechanism for predicting macro instructions and taking the appropriate action.

Instruction decode unit **402** decodes instructions received from the instruction cache **202** and provides the decoded instructions to the execute units **448**, the load/store unit **450**, or the DSP unit **214**. The instruction decode unit **402** is preferably configured to dispatch an instruction to more than one execute unit **448**.

The instruction decode unit **402** includes function preprocessor **204**. According to the first embodiment of the present invention, the function preprocessor **204** in the instruction decode unit **402** is configured to examine the status of the processor mode bit **213** to determine whether an X86 instruction sequence in the instruction cache **202** corresponds to or performs DSP functions. If the processor mode bit **213** is set to indicate such an instruction sequence, the function preprocessor **204** generates a corresponding macro and parameters and transmits the corresponding DSP macro and parameters to the DSP Unit **214** upon DSP dispatch bus **456**. The DSP unit **214** receives the DSP function macro and parameter information from the instruction decode unit **402** and performs the indicated DSP function. Additionally, DSP unit **214** is preferably configured to access data cache **444** for data operands. Data operands may be stored in a memory within DSP unit **214** for quicker access, or may be accessed directly from data cache **444** when needed. Function preprocessor **204** provides feedback to instruction cache **202** to ensure that sufficient look ahead instructions are available for macro searching.

If the processor mode bit **213** indicates that the X86 instructions in the instruction cache **202** are not intended to perform a DSP function, the instruction decode unit **402** decodes the instructions fetched from instruction cache **202** and dispatches the instructions to execute units **448** and/or load/store unit **450**. Instruction decode unit **402** also detects the register operands used by the instruction and requests these operands from reorder buffer **452** and register file **454**. Execute units **448** execute the X86 instructions as is known in the art

Also, if the DSP **214** is not included in the CPU **102** or is disabled through software, instruction decode unit **402** dispatches all X86 instructions to execute units **448**. Execute units **448** execute the X86 instructions as in the prior art. In this manner, if the DSP unit **214** is disabled, the X86 code, including the instructions which perform DSP functions, are executed by the X86 core, as is currently done in prior art X86 microprocessors. Thus, if the DSP unit **214** is disabled, the program executes correctly even though operation is less efficient than the execution of a corresponding routine in the DSP **214**. Advantageously, the enabling or disabling, or the presence or absence, of the DSP core **214** in the CPU **102** does not affect the correct operation of the program.

In one embodiment, execute units **448** are symmetrical execution units that are each configured to execute the instruction set employed by microprocessor **102**. In another embodiment, execute units **448** are asymmetrical execution units configured to execute dissimilar instruction subsets. For example, execute units **448** may include a branch execute unit for executing branch instructions, one or more arithmetic/logic units for executing arithmetic and logical instructions, and one or more floating point units for executing floating point instructions. Instruction decode unit **402** dispatches an instruction to an execute unit **448** or load/store unit **450** which is configured to execute that instruction.

Load/store unit **450** provides an interface between execute units **448** and data cache **444**. Load and store memory operations are performed by load/store unit **450** to data cache **444**. Additionally, memory dependencies between load and store memory operations are detected and handled by load/store unit **450**.

Execute units **448** and load/store unit(s) **450** may include one or more reservation stations for storing instructions whose operands have not yet been provided. An instruction is selected from those stored in the reservation stations for execution if: (1) the operands of the instruction have been provided, and (2) the instructions which are prior to the instruction being selected have not yet received operands. It is noted that a centralized reservation station may be included instead of separate reservations stations. The centralized reservation station is coupled between instruction decode unit **402**, execute units **448**, and load/store unit **450**. Such an embodiment may perform the dispatch function within the centralized reservation station.

CPU **102** preferably supports out of order execution and employs reorder buffer **452** for storing execution results of speculatively executed instructions and storing these results into register file **454** in program order, for performing dependency checking and register renaming, and for providing for mispredicted branch and exception recovery. When an instruction is decoded by instruction decode unit **402**, requests for register operands are conveyed to reorder buffer **452** and register file **454**. In response to the register operand requests, one of three values is transferred to the execute unit **448** and/or load/store unit **450** which receives the instruction: (1) the value stored in reorder buffer **452**, if the value has been speculatively generated; (2) a tag identifying a location within reorder buffer **452** which will store the result, if the value has not been speculatively generated; or (3) the value stored in the register within register file **454**, if no instructions within reorder buffer **452** modify the register. Additionally, a storage location within reorder buffer **452** is allocated for storing the results of the instruction being decoded by instruction decode unit **402**. The storage location is identified by a tag, which is conveyed to the unit receiving the instruction. It is noted that, if more than one reorder buffer storage location is allocated for storing results corresponding to a particular register, the value or tag corresponding to the last result in program order is conveyed in response to a register operand request for that particular register.

When execute units **448** or load/store unit **450** execute an instruction, the tag assigned to the instruction by reorder buffer **452** is conveyed upon result bus **458** along with the result of the instruction. Reorder buffer **452** stores the result in the indicated storage location. Additionally, execute units **448** and load/store unit **450** compare the tags conveyed upon result bus **458** with tags of operands for instructions stored therein. If a match occurs, the unit captures the result from result bus **458** and stores it with the corresponding instruction. In this manner, an instruction may receive the operands it is intended to operate upon. Capturing results from result bus **458** for use by instructions is referred to as "result forwarding".

Instruction results are stored into register file **454** by reorder buffer **452** in program order. Storing the results of an instruction and deleting the instruction from reorder buffer **452** is referred to as "retiring" the instruction. By retiring the instructions in program order, recovery from incorrect speculative execution may be performed. For example, if an instruction is subsequent to a branch instruction whose taken/not taken prediction is incorrect, then the instruction may be executed incorrectly. When a mispredicted branch instruction or an instruction which causes an exception is detected, reorder buffer **452** discards the instructions subsequent to the mispredicted branch instructions. Instructions thus discarded are also flushed from execute units **448**, load/store unit **450**, and instruction decode unit **402**.

Register file **454** includes storage locations for each register defined by the microprocessor architecture employed by microprocessor **102**. For example, in the preferred embodiment where the CPU **102** includes an x86 microprocessor architecture, the register file **454** includes locations for storing the EAX, EBX, ECX, EDX, ESI, EDI, ESP, and EBP register values.

Data cache **444** is a high speed cache memory configured to store data to be operated upon by microprocessor **102**. It is noted that data cache **444** may be configured into a set-associative or direct-mapped configuration.

For more information regarding the design and operation of an X86 compatible microprocessor, please see co-pending patent application entitled "High Performance Superscalar Microprocessor", Ser. No. 08/146,382, filed Oct. 29, 1993 by Witt, et al, now U.S. Pat. No. 5,651,125, and co-pending patent application entitled "Superscalar Microprocessor Including a High Performance Instruction Alignment Unit", Ser. No. 08/377,843, filed Jan. 25, 1995 by Witt, et al now U.S. Pat. No. 5,819,057, which are both assigned to the assignee of the present application, and which are both hereby incorporated by reference in their entirety as though fully and completely set forth herein. Please also see "Superscalar Microprocessor Design" by Mike Johnson, Prentice-Hall, Englewood Cliffs, N.J., 1991, which is hereby incorporated herein by reference in its entirety.

FIG. 5—Instruction Decode Unit

Referring now to FIG. 5, one embodiment of instruction decode unit **402** is shown. Instruction decode unit **402** includes an instruction alignment unit **460**, a plurality of decoder circuits **462**, processor mode register or bit **213**, and a DSP function preprocessor **204**. Instruction alignment unit **460** is coupled to receive instructions fetched from instruction cache **202** and aligns instructions to decoder circuits **462**.

Instruction alignment unit **260** routes instructions to decoder circuits **462**. In one embodiment, instruction alignment unit **260** includes a byte queue in which instruction bytes fetched from instruction cache **202** are queued. Instruction alignment unit **460** locates valid instructions from within the byte queue and dispatches the instructions to respective decoder circuits **462**. In another embodiment, instruction cache **202** includes predecode circuitry which predecodes instruction bytes as they are stored into instruction cache **202**. Start and end byte information indicative of the beginning and end of instructions is generated and stored within instruction cache **202**. The predecode data is transferred to instruction alignment unit **460** along with the instructions, and instruction alignment unit **460** transfers instructions to the decoder circuits **462** according to the predecode information.

The function preprocessor **204** is also coupled to the instruction cache **202**. As described above, the function preprocessor **204** examines the processor mode bit in order to detect instruction sequences in the instruction cache **202** which perform DSP instructions. Decoder circuits **462** and function preprocessor **204** receive X86 instructions from the instruction alignment unit **460**. The function preprocessor **204** provides an instruction disable signal upon a DSP bus to each of the decoder units **462**.

Each decoder circuit **462** decodes the instruction received from instruction alignment unit **460** to determine the register operands manipulated by the instruction as well as the unit to receive the instruction. An indication of the unit to receive the instruction as well as the instruction itself are conveyed upon a plurality of dispatch buses **468** to execute units **448** and load/store unit **450**. Other buses, not shown, are used to request register operands from reorder buffer **452** and register file **454**.

The function preprocessor **204** examines the processor mode bit to determine if streams or sequences of X86 instructions from the instruction cache **202** implement a DSP function. If so, the function preprocessor **204** maps the X86 instruction stream to a DSP macro and zero or more parameters and provides this information to one of the one or more DSP units **214**. In one embodiment, when the respective instruction sequence reaches the decoder circuits **462**, the function preprocessor **204** asserts a disable signal to each of the decoders **462** to disable operation of the decoders **462** for the detected instruction sequence. When a decoder circuit **462** detects the disable signal from function preprocessor **204**, the decoder circuit **462** discontinues decoding operations until the disable signal is released. After the instruction sequence corresponding to the DSP function has exited the instruction cache **202**, the processor mode bit is cleared, and the function preprocessor **204** removes the disable signal to each of the decoders **462**. In other words, once the processor mode bit is cleared and the function preprocessor **204** detects the end of the X86 instruction sequence, the function preprocessor **204** removes the disable signal to each of the decoders **462**, and the decoders resume operation.

Each of decoder circuits **462** is configured to convey an instruction upon one of dispatch buses **468**, along with an indication of the unit or units to receive the instruction. In one embodiment, a bit is included within the indication for each of execute units **448** and load/store unit **450**. If a particular bit is set, the corresponding unit is to execute the instruction. If a particular instruction is to be executed by more than one unit, more than one bit in the indication may be set.

Function Preprocessor

As shown in FIG. **5**, in the first embodiment the function preprocessor **204** comprises a conversion/mapping circuit **506** for converting a sequence of instructions in the instruction memory **202** which implements a digital signal processing function into a digital signal processing function identifier or macro identifier and zero or more parameters. Thus if the processor mode bit indicates that the sequence of instructions in the instruction memory **202** implements a DSP function, the conversion/mapping circuit **506** converts this sequence of instructions into a DSP function identifier and zero or more parameters. For example, if the instruction sequence determination circuit **504** examines and determines that the sequence of instructions in the instruction memory **202** implements an FFT function, the conversion/mapping circuit **506** converts this sequence of instructions into a FFT function identifier and zero or more parameters.

As discussed above with respect to step **312** of FIG. **3**, in one embodiment of the invention the processor mode register **213** stores a processor mode bit and in addition stores one or more bits, preferably a plurality of bits, which indicate the general type of DSP function being performed. Thus the application program writes a value into the processor mode register **213** indicating the type of DSP function being implemented by the APX instruction sequence. The conversion/mapping circuit **506** uses the value indicating the type of DSP function to aid in converting the sequence of instructions into a DSP function identifier and zero or more parameters.

FIG. 6—Pattern Recognition Circuit

Referring now to FIG. **6**, in one embodiment the function preprocessor **204** includes a pattern recognition circuit or pattern recognition detector **512** which determines whether a sequence of instructions in the instruction memory **202** implements a digital signal processing function. The pattern recognition circuit **512** is used to convert the sequence of instructions into a DSP function identifier and zero or more parameters.

The pattern recognition circuit **512** stores a plurality of patterns of instruction sequences which implement digital signal processing functions. The pattern recognition circuit **512** stores bit patterns which correspond to opcode sequences of machine language instructions which perform DSP functions, such as FFTs, inner products, matrix manipulation, correlation, convolution, etc.

For instruction sequences where the processor mode bit is set to indicate that the sequence implements a DSP function, the pattern recognition detector **512** compares each of the patterns with the respective instruction sequence. The pattern recognition detector **512** examines the sequence of instructions stored in the instruction memory **202** and compares the sequence of instructions with the plurality of stored patterns. Operation of the pattern recognition detector **512** is shown in FIG. **7**. The pattern recognition detector **512** may include a look-up table as the unit which performs the pattern comparisons, as desired. The pattern recognition detector **512** may also perform macro prediction on instruction sequences to improve performance.

The pattern recognition detector **512** determines whether the sequence of instructions in the instruction memory **202** substantially matches one of the plurality of stored patterns. A substantial match indicates that the sequence of instructions implements the respective digital signal processing function. In the preferred embodiment, a substantial match occurs where the instruction sequence matches a stored pattern by greater than 90%. Other matching thresholds, such as 95%, or 100%, may be used, as desired. The pattern recognition detector **512** determines the type of DSP function pattern which matched the sequence of instructions and passes this DSP function type to the conversion/mapping circuit **506**.

FIG. 8—Look-up Table

Referring now to FIG. **8**, in another embodiment the conversion/mapping circuit **506** includes a look-up table (LUT) **514** which determines the digital signal processing function that corresponds to a sequence of instructions in the instruction memory **202**. In this embodiment, the look-up table **514** may be in addition to, or instead of, the pattern recognition detector **512**. Thus the LUT **514** is used in converting the sequence of instructions into a DSP function identifier and zero or more parameters. The LUT operates as shown in FIG. **9**.

In an embodiment where the function preprocessor **204** includes only the look-up table **514**, the look-up table **514** stores a plurality of patterns wherein each of the patterns is at least a subset of an instruction sequence which implements a digital signal processing function. Thus, this embodiment is similar to the embodiment of FIG. **6** described above, except that the function preprocessor **204** includes the look-up table **514** instead of the pattern recognition detector **512** for determining which DSP function corresponds to an instruction sequence. In this embodiment, the look-up table **514** requires an exact match with a corresponding sequence of instructions. If an exact match does not occur, then the sequence of instructions are passed to the one or more general purpose execution units, i.e., the general purpose CPU core, for execution.

FIG. **9** illustrates operation of the look-up table **514** in this embodiment. As shown, a sequence of instructions in the instruction cache **202** are temporarily stored in the instruction latch **542**. If the processor mode bit indicates that the instruction sequence implements a DSP function, then the contents of the instruction latch **542** are then compared with each of the entries in the look-up table **514** by element **546**. If the contents of the instruction latch **542** exactly match one of the entries in the look-up table **514**, then the DSP function or instruction **548** which corresponds to this entry is provided to the DSP execution unit **214**.

In the above embodiments of FIGS. **6** and **8**, the pattern recognition detector **512** and/or the look-up table **514** are configured to determine the DSP function which corresponds to an instruction sequence only when the determination can be made with relative certainty. This is because a "missed" instruction sequence, i.e., an instruction sequence which implements a DSP function, wherein the type of DSP instruction could not be positively identified, will not affect operation of the CPU **102**, since the general purpose core or execution units can execute the instruction sequence. However, an instruction sequence which does implements a DSP function that is mis-identified, i.e., the wrong DSP function is determined to be implemented, is more problematic, and could result in possible erroneous operation. Thus it is anticipated that the pattern recognition detector **512** or the look-up table **514** may not accurately detect every instruction sequence which implements a DSP function. In this instance, even though the processor mode bit was set to indicate that the instruction sequence implemented a DSP function, the instruction sequence is preferably passed on to one of the general purpose execution units, as occurs in the prior art.

FIG. 10—Second Embodiment

FIG. **10** is a high level block diagram of the CPU **102** according to the second embodiment of the invention. Thus, FIG. **10** is similar to FIG. **2**, but illustrates the second embodiment described above.

As shown, the CPU **102** includes an instruction cache or instruction memory **202** which receives instructions or opcodes from the system memory **110**. In this second embodiment, the instructions comprise sequences of x86 or APX instructions and sequences of DSP instructions. Thus, unlike the first embodiment of FIG. **2** wherein all received instructions were APX instructions, in this second embodiment the received instructions comprises APX instruction sequences and DSP instruction sequences.

Preprocessor **204A** is coupled to the instruction memory **202** and examines instruction sequences or opcode sequences in the instruction memory **202**. The preprocessor **204A** is also coupled to the X86 core **212** and the DSP core **214**. The function preprocessor **204A** is further coupled to the processor mode register **213** storing the processor mode bit. As shown, the preprocessor **204A** examines the processor mode bit and selectively provides APX instructions or opcodes to the X86 core **212** or selectively provides DSP op-codes or instructions to the DSP core **214**.

The X86 core **212** and DSP core **214** are coupled together and provide data and timing signals between each other. In one embodiment, the CPU **102** includes one or more buffers (not shown) which interface between the X86 core **212** and the DSP core **214** to facilitate transmission of data between the X86 core **212** and the DSP core **214**.

In this second embodiment, the CPU **212** receives instructions which comprise sequences of general purpose, e.g., APX instructions, and which also comprises sequences of DSP instructions. The respective processor mode bit is set to indicate the beginning of a sequence of DSP instructions, and the processor mode bit is cleared to indicate the beginning of a sequence of APX instructions. The preprocessor **204A** thus routes the instructions to the APX core or the DSP core based on the status of the processor mode bit. In this embodiment, the pre-processor **204A** is not required to map APX instructions into DSP macros, but rather simply routes APX instructions to the x 86 core **212** and routes DSP instructions to the DSP core **214** based on the status of the processor mode bit.

FIG. **11**—Flowchart Diagram: Second Embodiment

FIG. **11** is a flowchart diagram illustrating the second embodiment. As described above, in this second embodiment the CPU **102** receives an instruction sequence which comprises sequences of general purpose, e.g., APX instructions, and which also comprises sequences of DSP instructions. The respective processor mode bit is set to indicate the beginning of a sequence of DSP instructions, and the processor mode bit is cleared to indicate the beginning of a sequence of APX instructions. The CPU **102** thus routes the instructions to the APX core or the DSP core based on the status of the processor mode bit.

As shown, in step **802** the CPU **102** receives sequences of instructions. As noted above, these instructions comprise sequences of general purpose, e.g., APX instructions, and also comprise sequences of DSP instructions. In step **804** the preprocessor **204** examines the processor mode bit to determine if a respective sequence is a sequence of APX instructions or a sequence of DSP instructions.

In step **806** the preprocessor **204A** determines, based on the status of the processor mode bit, if the respective sequence is a sequence of APX instructions or a sequence of DSP instructions. If the processor mode bit is cleared to indicate that the instructions or opcodes stored in the instruction cache **202** are not DSP instructions, the instructions are provided to the X86 core **212** in step **808**. Thus, these instructions or opcodes are provided directly from the instruction cache **202** to the X86 core **212** for execution, as occurs in prior art X86 compatible CPUs. After the opcodes are transferred to the X86 core **212**, in step **810** the X86 core **212** executes the instructions.

If the processor mode bit is set to indicate that the sequence of instructions comprise DSP instructions in step **806**, then in step **812** the preprocessor **204A** provides the DSP instruction sequence to the DSP core **214**. In step **314** the DSP core **214** executes the DSP instructions.

FIG. 12—Processor Mode Register

FIG. **12** illustrates one embodiment of the processor mode register **213**. As shown, in one embodiment, a special register in the APX CPU includes one or more bits, referred to as processor mode bits, assigned to indicate the processor mode, i.e., which indicate whether an instruction sequence comprises DSP instructions or implements a DSP function, or whether the instruction sequence is a regular APX instruction sequence.

FIG. **13**—Instruction Sequence

FIG. **13** illustrates one embodiment of an instruction sequence which includes a DSP instruction sequence. As shown, after a number of APX instructions, e.g. three instructions, a DSP routine is called. The DSP routine sets the DSP bit to indicate the start of a sequence of DSP instructions. After the DSP instructions or operations are executed by the DSP core **214**, the routine clears the DSP bit and returns to execution of APX instructions.

Conclusion

Therefore, the present invention comprises a novel CPU or microprocessor architecture which optimizes execution of DSP and/or mathematical operations while maintaining backwards compatibility with existing software.

Although the system and method of the present invention has been described in connection with the preferred embodiment, it is not intended to be limited to the specific form set forth herein, but on the contrary, it is intended to cover such alternatives, modifications, and equivalents, as can be reasonably included within the spirit and scope of the invention as defined by the appended claims.

We claim:

1. A central processing unit which performs general purpose processing functions and digital signal processing (DSP) functions, comprising:

an instruction memory for storing a plurality of instructions, wherein said instruction memory stores one or more sequences of instructions which are intended to perform a DSP function;

a processor mode memory for storing one or more processor mode bits, wherein said one or more processor mode bits indicate whether a sequence of instructions implements a DSP function;

a function preprocessor coupled to the instruction memory and coupled to the processor mode memory, wherein the function preprocessor is operable to examine said one or more processor mode bits in said processor mode memory to determine whether a sequence of said instructions in said instruction memory is intended to perform a digital signal processing function, wherein the function preprocessor is operable to convert said sequence of said instructions in said instruction memory into a DSP function identifier if said one or more processor mode bits in said processor mode memory indicate that said sequence of said instructions in said instruction memory is intended to perform a DSP function;

at least one general purpose processing core coupled to the function preprocessor for executing instructions in said instruction memory, wherein the function preprocessor provides a sequence of instructions to said at least one general purpose processing core if said one or more processor mode bits indicate that said sequence of said instructions in said instruction memory is not intended to perform a DSP function;

at least one digital signal processing core coupled to the function preprocessor for performing digital signal processing functions, wherein the function preprocessor is operable to provide said digital signal processing function identifier to said at least one digital signal processing core, wherein the at least one digital signal processing core receives said digital signal processing function identifier and performs a digital signal processing function in response to said received digital signal processing function identifier from said function preprocessor.

2. The central processing unit of claim 1, wherein said instruction memory stores a first sequence of instructions which does not perform a digital signal processing function, and wherein said instruction memory stores a second sequence of instructions which performs a digital signal processing function;

wherein said at least one general purpose processing core executes said first sequence of instructions;

wherein said at least one digital signal processing core performs said digital signal processing function in response to said received digital signal processing function identifier, wherein said digital signal processing function performed by said digital signal processing core is substantially equivalent to execution of said second sequence of instructions.

3. The central processing unit of claim 1, wherein said processor mode memory stores a respective value for said one or more processor mode bits for each respective sequence of instructions in said instruction memory;

wherein said respective value indicates whether said respective sequence of instructions implements a DSP function.

4. The central processing unit of claim 1, wherein said processor mode memory stores a value indicating a type of DSP function implemented by a sequence of instructions;

wherein said processor mode memory stores said value indicating said type of DSP function implemented by said sequence of instructions when said one or more processor mode bits indicate that said sequence of instructions implements a DSP function;

wherein said function preprocessor uses said value indicating said type of DSP function implemented by said sequence of instructions in converting said sequence of said instructions in said instruction memory into a DSP function identifier.

5. The central processing unit of claim 1, wherein said at least one digital signal processing core provides data and timing signals to said at least one general purpose processing core.

6. The central processing unit of claim 1, wherein said function preprocessor generates a digital signal processing function identifier and one or more parameters in response to said one or more processor mode bits indicating that said sequence of instructions in said instruction memory is intended to perform a digital signal processing function.

7. The central processing unit of claim 1, wherein said at least one general purpose processing core is compatible with the X86 family of microprocessors.

8. The central processing unit of claim 7, wherein said plurality of instructions are X86 opcodes.

9. The central processing unit of claim 1, wherein said at least one digital signal processing core is adapted for performing one or more mathematical operations from the group consisting of convolution, correlation, Fast Fourier Transforms, and inner product.

10. The central processing unit of claim 1, wherein said at least one general purpose processing core and said at least one digital signal processing core operate substantially in parallel.

11. A method for executing instructions in a central processing unit (CPU), wherein the CPU includes at least one general purpose CPU core and at least one digital signal processing (DSP) core, the method comprising:

storing one or more sequences of instructions in an instruction memory for execution by the central processing unit;

storing one or more processor mode bits in a processor mode memory, wherein said one or more processor mode bits indicate whether a sequence of instructions implements a DSP function;

examining a sequence of instructions in said instruction memory;

examining said one or more processor mode bits to determine whether said sequence of instructions in said instruction memory is intended to perform a DSP function;

converting said sequence of instructions in said instruction memory into a DSP function identifier if said one or more processor mode bits indicate that said sequence of instructions in said instruction memory is intended to perform a DSP function;

the digital signal processing core receiving said DSP function identifier;

the digital signal processing core performing a digital signal processing function in response to said received digital signal processing function identifier.

12. The method of claim 11, further comprising:

said general purpose central processing unit core executing said sequence of instructions if said one or more processor mode bits indicate that said sequence of instructions in said instruction memory is not intended to perform a DSP function.

13. The method of claim 12, further comprising:

wherein said storing comprises storing a first sequence of instructions in said instruction memory which performs a first digital signal processing function;

wherein said storing comprises storing a second sequence of instructions in said instruction memory which does not perform a digital signal processing function;

wherein said converting converts said first sequence of instructions in said instruction memory which is intended to perform said first digital signal processing function into a first digital signal processing function identifier;

wherein said performing comprises said digital signal processing core performing said first digital signal processing function in response to said first digital signal processing function identifier, wherein said performing said first digital signal processing function is substantially equivalent to execution of said first sequence of instructions; and

said general purpose central processing unit core executing said second sequence of instructions.

14. The method of claim 11, wherein said storing one or more processor mode bits in the processor mode memory comprises storing a respective value for said one or more processor mode bits for each respective sequence of instructions in said instruction memory;

wherein said respective value indicates whether said respective sequence of instructions implements a DSP function.

15. The method of claim 11, further comprising:

storing a value in said processor mode memory indicating a type of DSP function implemented by a sequence of instructions;

wherein said processor mode memory stores said value indicating said type of DSP function implemented by said sequence of instructions when said one or more processor mode bits indicate that said sequence of instructions implements a DSP function;

wherein said function preprocessor uses said value indicating said type of DSP function implemented by said sequence of instructions in converting said sequence of said instructions in said instruction memory into a DSP function identifier.

16. The method of claim 11, further comprising:

said digital signal processing core and said general purpose central processing unit core operating substantially in parallel.

17. The method of claim 11, further comprising:

said digital signal processing core providing data and timing signals to said general purpose central processing unit core.

18. The method of claim 11, further comprising:

said function preprocessor generating a digital signal processing function identifier and one or more parameters in response to said determining that said sequence of instructions in said instruction memory is intended to perform a digital signal processing function.

19. The method of claim 9, wherein said general purpose central processing unit core is compatible with the X86 family of microprocessors;

wherein said one or more sequences of instructions comprise X86 opcodes.

20. The method of claim 11, wherein said digital signal processing core performs one or more mathematical operations from the group consisting of convolution, correlation, Fast Fourier Transform, and inner product.

* * * * *

US006247121B1

(12) **United States Patent**
Akkary et al.

(10) **Patent No.: US 6,247,121 B1**
(45) **Date of Patent: Jun. 12, 2001**

(54) **MULTITHREADING PROCESSOR WITH THREAD PREDICTOR**

(75) Inventors: **Haitham Akkary**, Portland, OR (US); **Quinn A. Jacobson**, Madison, WI (US)

(73) Assignee: **Intel Corporation**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/157,791**

(22) Filed: **Sep. 21, 1998**

Related U.S. Application Data

(63) Continuation-in-part of application No. 08/992,375, filed on Dec. 16, 1997.

(51) **Int. Cl.**^{7} **C06F 15/00**

(52) **U.S. Cl.** **712/239**; 712/32; 712/216; 712/228; 712/233; 712/237; 712/238; 712/239; 712/240; 709/112

(58) **Field of Search** 712/32, 216, 228, 712/240, 233, 234, 235, 236, 237, 238, 239; 709/112

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,142,634	8/1992	Fite et al.	712/240
5,153,848	10/1992	Elkind et al.	708/503
5,309,561	5/1994	Overhouse et al.	709/400
5,313,634	5/1994	Eickemeyer	712/240
5,420,990	5/1995	McKeen et al.	712/216
5,421,021	5/1995	Saini	712/211
5,524,250	6/1996	Chesson et al.	712/228
5,524,262	6/1996	Colwell et al.	712/213
5,546,593	8/1996	Kimura et al.	712/228
5,586,278	12/1996	Papworth et al.	712/235
5,588,126	12/1996	Abramson et al.	712/200
5,606,670	2/1997	Abramson et al.	711/154
5,613,083	3/1997	Glew et al.	711/207

(List continued on next page.)

OTHER PUBLICATIONS

M. Franklin, "The Multiscalar Architecture," Ph.D. Dissertation, Univ. of Wisconsin, 1993, pp. i, ii, v–ix, 50–73, 75–81, 86–107, and 153–161.

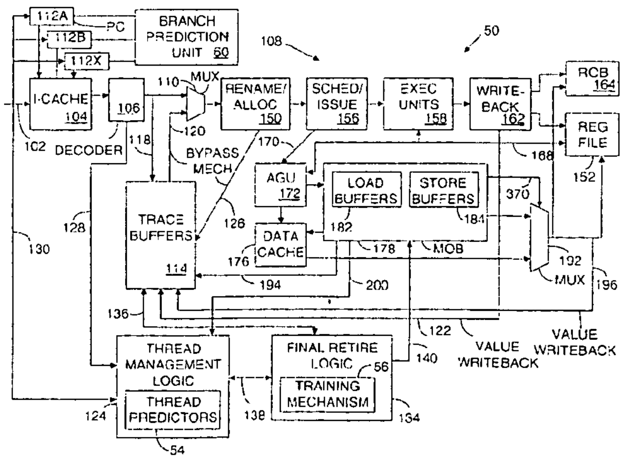
(List continued on next page.)

Primary Examiner—John A. Follansbee
Assistant Examiner—Dzung C Nguyen
(74) *Attorney, Agent, or Firm*—Alan K. Aldous

(57) **ABSTRACT**

In one embodiment, a processor includes thread management logic including a thread predictor having state machines to indicate whether thread creation opportunities should be taken or not taken. The processor includes a predictor training mechanism to receive retired instructions and to identify potential threads from the retired instructions and to determine whether a potential thread of interest meets a test of thread goodness, and if the test is met, one of the state machines that is associated with the potential thread of interest is updated in a take direction, and if the test is not met, the state machine is updated in a not take direction. The thread management logic may control creation of an actual thread and may further include reset logic to control whether the actual thread is reset and wherein if the actual thread is reset, one of the state machines associated with the actual thread is updated in a not take direction. The final retirement logic may control whether the actual thread is retired, and wherein if the actual thread is retired, the state machine associated with the actual thread is updated in a take direction. The circuitry may be used in connection with a multi-threading processor that detects speculation errors involving thread dependencies in execution of the actual threads and re-executes instructions associated with the speculation errors from trace buffers outside an execution pipeline.

22 Claims, 10 Drawing Sheets



U.S. PATENT DOCUMENTS

5,664,137	9/1997	Abramson et al.	712/216
5,724,565	3/1998	Dubey et al.	712/245
5,742,782	4/1998	Ito et al.	712/245
5,754,818	5/1998	Mohamed	711/207
5,802,272	9/1998	Sites et al.	714/45
5,812,811	9/1998	Dubey et al.	712/216
5,832,260	11/1998	Arora et al.	712/239
5,881,280	3/1999	Gupta et al.	712/244
5,887,166	3/1999	Mallick et al.	709/112
5,933,627	8/1999	Parady	711/141
5,961,639	10/1999	Mallick et al.	712/224

OTHER PUBLICATIONS

M. Franklin et al., ARB: A Hardware Mechanism for Dynamic Reordering of Memory References, IEEE Transactions on Computers, vol. 45, No. 5, May 1996, pp. 552–571.

E. Rotenberg et al., "Trace Processors," The 30th International Symposium on Microarchitecture, Dec. 1997, pp. 138–148.

J. Smith et al., "The Microarchitecture of Superscaler Processors," Proceedings of the IEEE, vol. 83, No. 12, Dec. 1995, pp. 1609–1624.

G. Sohi et al., "Multiscalar Processors". The 22nd Annual International Symposium on Computer Architecture, Jun. 1995, pp. 414–425.

P. Song, "Multithreading Comes of Age," Microprocessor Report, Jul. 14, 1997, pp. 13–18.

J. Tsai et al., "The Superthreaded Architecture: Thread Pipelining with Run–Time Data Dependence Checking and Control Speculation," Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques, Oct. 1996, pp. 35–46.

D. Tullsen et al., "Simultaneous Multithreading: Maximizing On–Chip Parallelism," The 22nd International Symposium on Computer Architecture, Jun. 1995, pp. 392–403.

U.S. application No. 08/746,547, filed Nov. 13, 1996, pending, "Processor Having Replay Architecture," Inventor David Sager.

J.C. Steffan et al., "The Potential for Using Thread–Level Data Speculation to Facilitate Automatic Parallelization," 4th International Symposium on High performance Computer Architecture—HPCA–4, pp. 2–13, Jan. 1998.

S. Gopal et al., "Speculative Versioning Cache," 4th International Symposium on High–Performance Computer Architecture—HPCA–4, pp. 195–205, Jan. 1998.

P. Marcuello et al., "Speculative Multithreading Processors," Proceedings of the ACM International Conference on Supercomputing 98, Jul. 1998, pp. 77–84.

N. Gloy et al., "An Analysis of Dynamic Branch Preduction Schemes on System Workloads," Proceedings, 23rd Annual International Symposium on Computer Architecture, May 1996, pp. 12–21.

S. McFarling, "Combining Branch Predictors," WRL Technical Note TN–36, Wetern Research Laboratory, Jun. 1993, pp. 1–20.

Q. Jacobson et al. "Path–Based Next Trace Prediction," Proceedings of the 30th International Symposium on Microarchitecture, Dec. 1997, pp. 14–23.

Q. Jacobson et al. "Control Flow Speculation in Multiscalar Processors," Proceedings of the 3rd International Symposium on High–Performance Computer Architecture, Feb. 1997, pp. 218–229.

R. Nair, "Dynamic path–based branch correlation," Proceedings of the 28th International Symposium on Microarchitecture, Dec. 1995, pp. 15–23.

S. Palacharla et al., "Complexity–Effective Superscalar Processors," The 24th Annual International Symposium, on Computer Architecture, pp. 206–218, Jun. 1997.

M. Lipasti et al., "Value Locality and Load Value Prediction," Proceedings of Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, Oct. 1996, ASPLOS–VII, pp. 138–147.

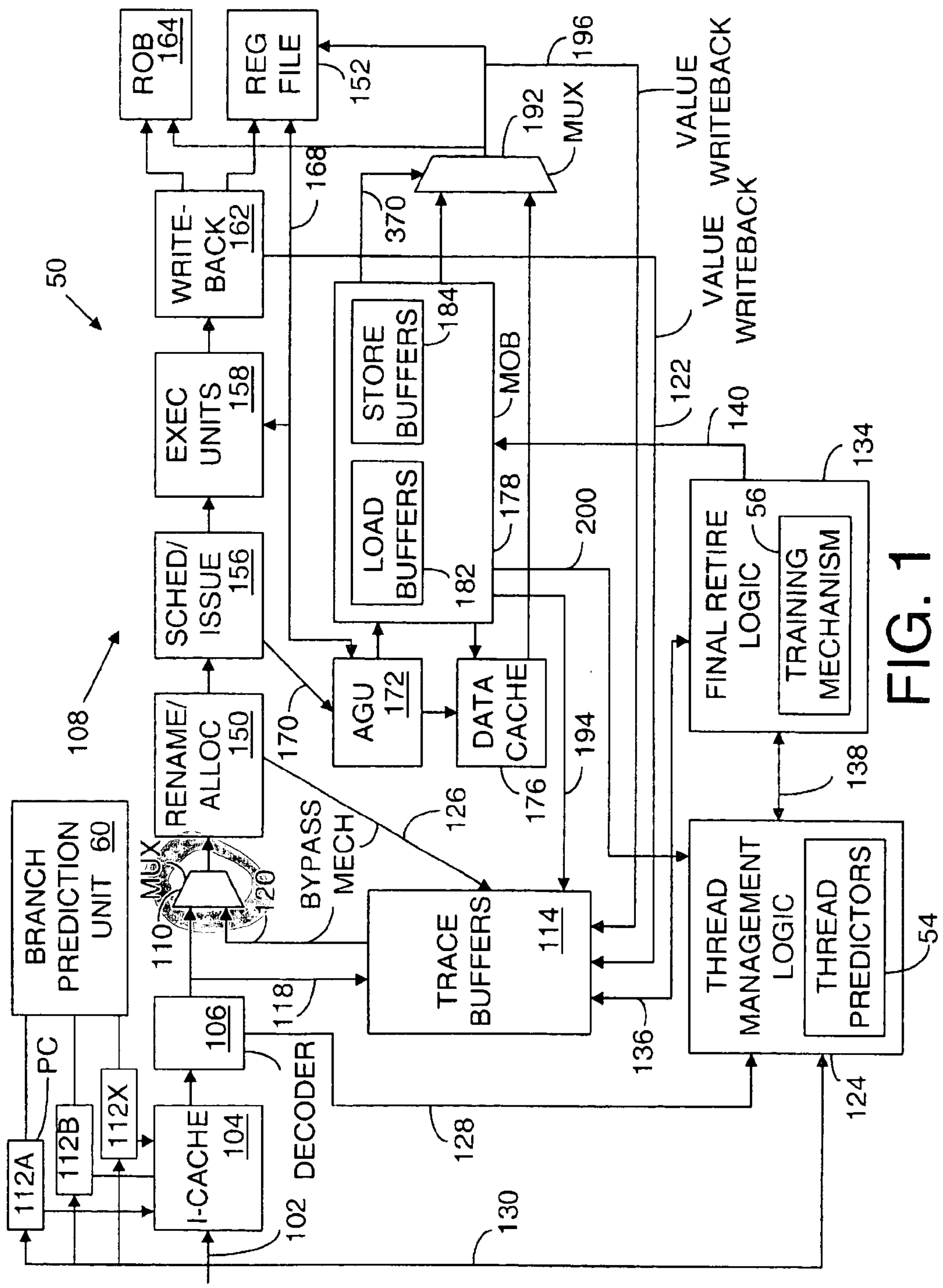

50
108
BRANCH PREDICTION UNIT 60
PC
112A
112B
112X
I-CACHE 104
102
DECODER
106
128
130
118
110
MUX
120
BYPASS MECH
126
RENAME/ ALLOC 150
SCHED/ ISSUE 156
EXEC UNITS 158
WRITE-BACK 162
ROB 164
REG FILE
152
168
170
AGU 172
DATA CACHE
176
194
TRACE BUFFERS 114
136
LOAD BUFFERS
182
STORE BUFFERS
184
MOB
178
200
370
192
MUX
196
VALUE WRITEBACK
VALUE WRITEBACK
122
140
FINAL RETIRE LOGIC
56
TRAINING MECHANISM
134
138
THREAD MANAGEMENT LOGIC
THREAD PREDICTORS
54
124

FIG. 1

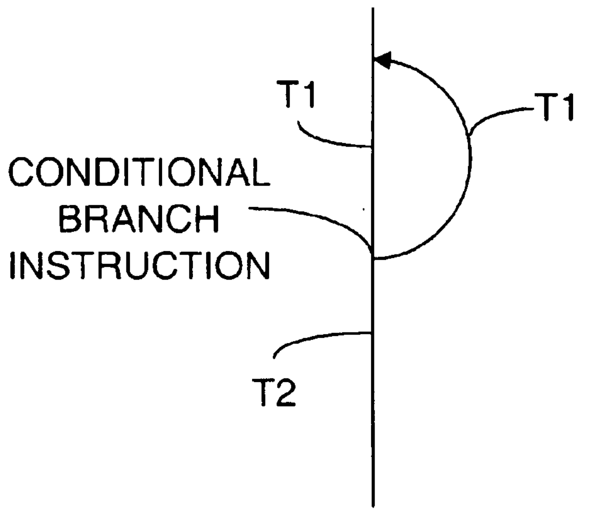

T1
T1
CONDITIONAL BRANCH INSTRUCTION
T2

FIG. 2

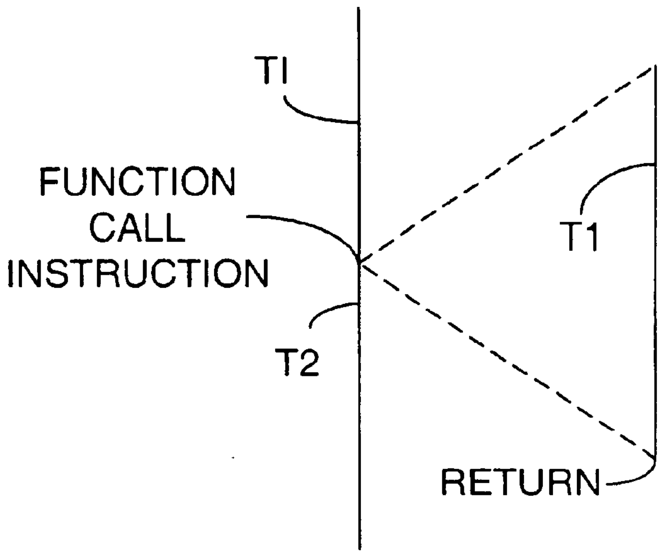

TI
FUNCTION CALL INSTRUCTION
T2
T1
RETURN

FIG. 3

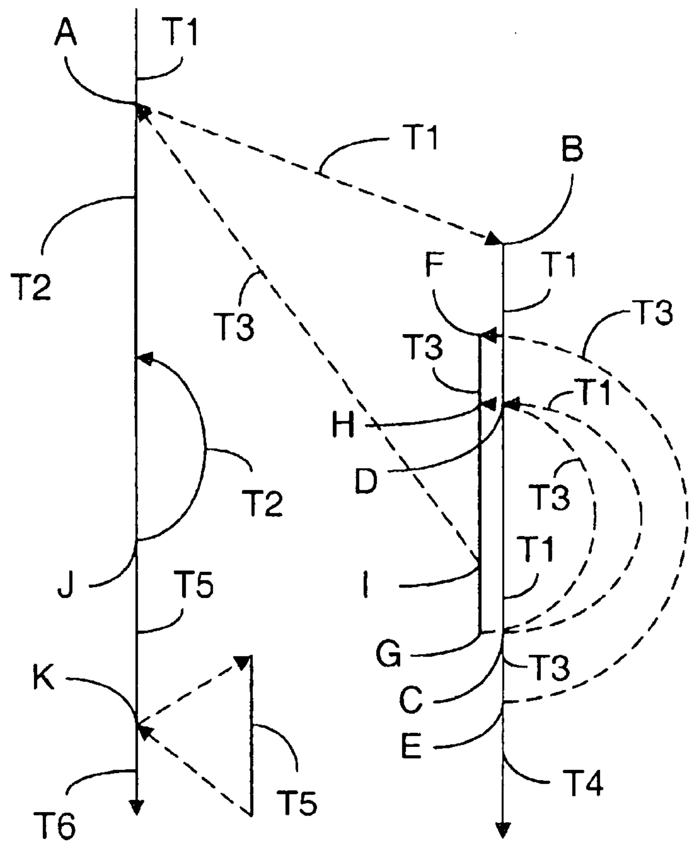

A
T1
T1
B
T2
T3
F
T1
T3
T3
T1
H
D
T2
T3
T1
J
T5
I
G
K
T3
C
E
T4
T5
T6

FIG. 4

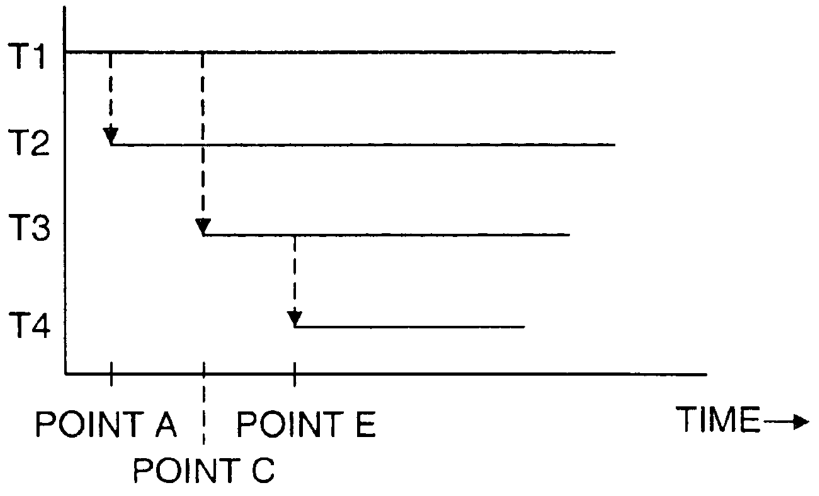

T1
T2
T3
T4
POINT A
POINT E
POINT C
TIME→

FIG. 5

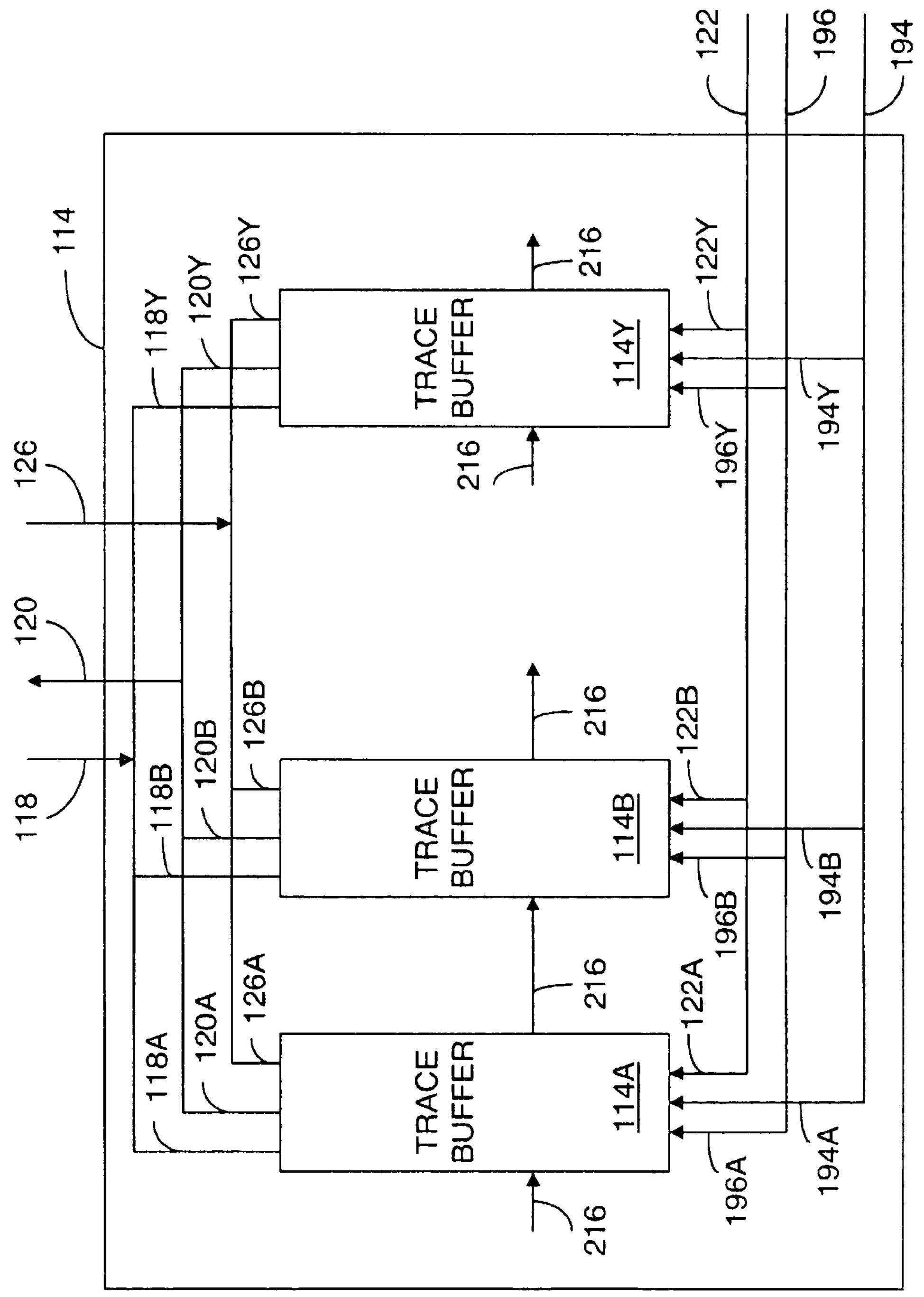

114
118
120
126
122
196
194
TRACE BUFFER
114A
TRACE BUFFER
114B
TRACE BUFFER
114Y
118A
120A
126A
118B
120B
126B
118Y
120Y
126Y
122A
196A
194A
122B
196B
194B
122Y
196Y
194Y
216

FIG. 6

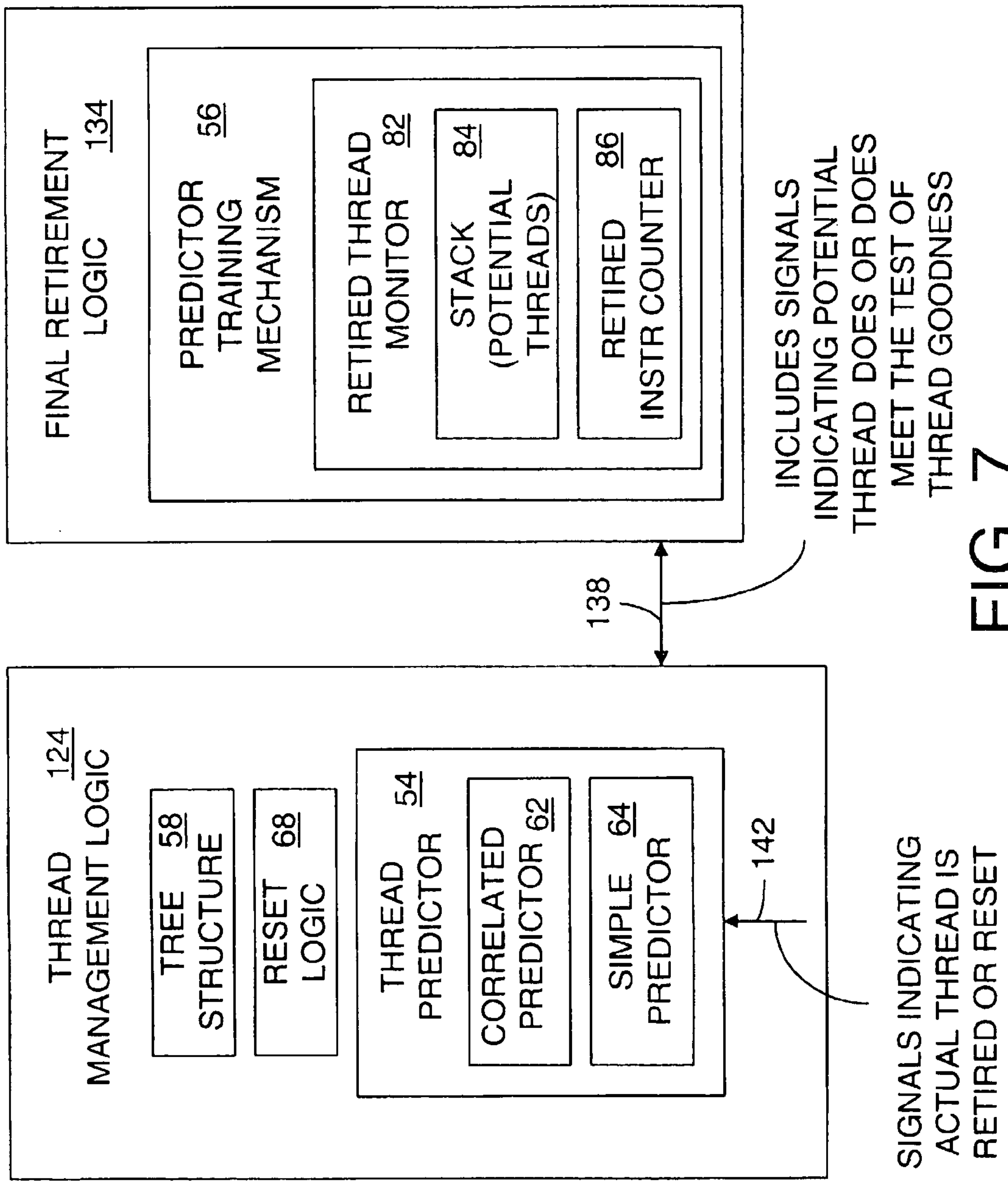
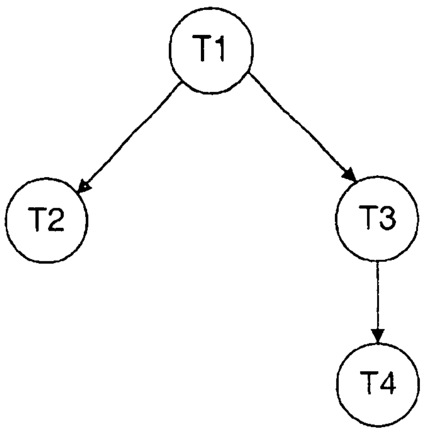
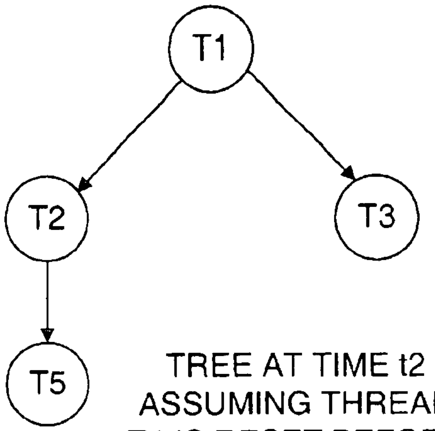

FINAL RETIREMENT LOGIC 134
PREDICTOR TRAINING MECHANISM 56
RETIRED THREAD MONITOR 82
STACK 84 (POTENTIAL THREADS)
RETIRED 86 INSTR COUNTER
INCLUDES SIGNALS INDICATING POTENTIAL THREAD DOES OR DOES MEET THE TEST OF THREAD GOODNESS
138
THREAD MANAGEMENT LOGIC 124
TREE 58 STRUCTURE
RESET 68 LOGIC
THREAD 54 PREDICTOR
CORRELATED PREDICTOR 62
SIMPLE 64 PREDICTOR
142
SIGNALS INDICATING ACTUAL THREAD IS RETIRED OR RESET

FIG. 7



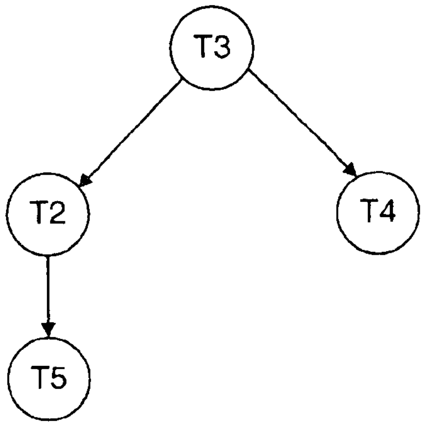
TREE AT TIME t1

FIG. 8A



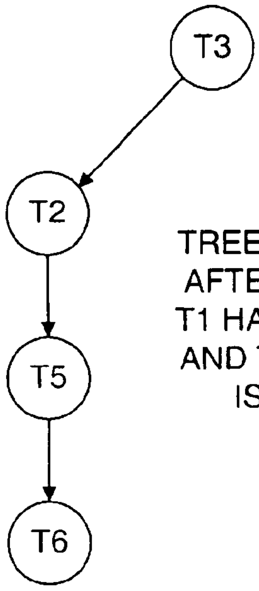
TREE AT TIME t2 ASSUMING THREAD T4 IS RESET BEFORE THREAD T1 RETIRES

FIG. 8B



TREE AT TIME t2 ASSUMING THREAD T1 RETIRES BEFORE THREAD T4 IS RESET

FIG. 8C



TREE AT TIME t3 AFTER THREAD T1 HAS RETIRED AND THREAD T4 IS RESET

FIG. 8D

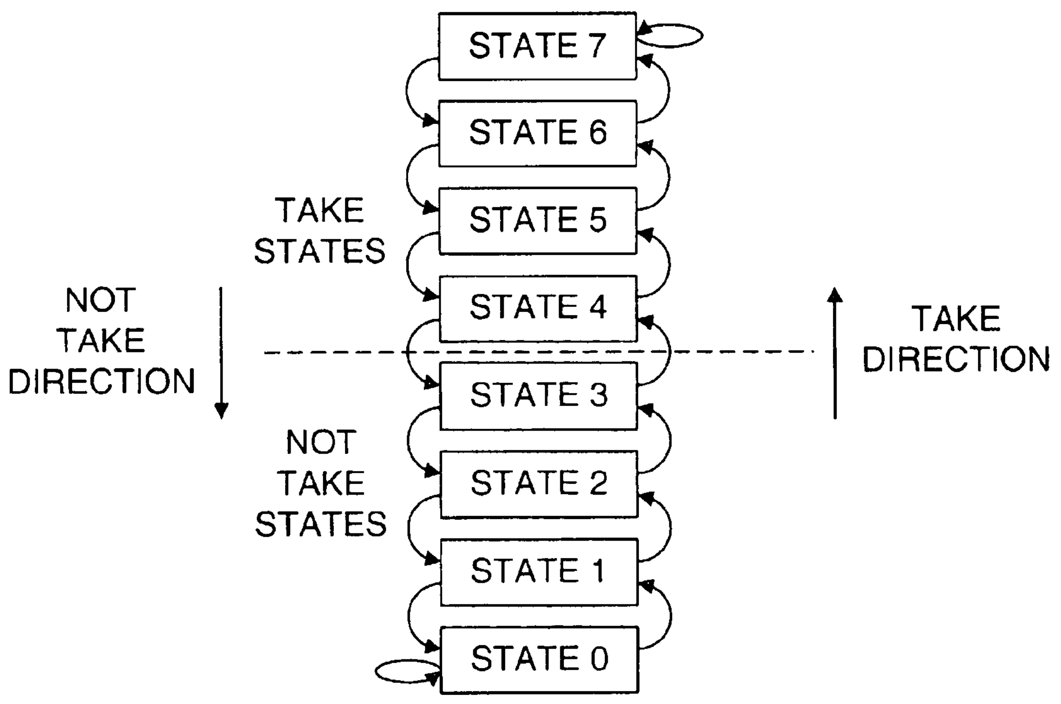

STATE 7
STATE 6
STATE 5
STATE 4
STATE 3
STATE 2
STATE 1
STATE 0
TAKE STATES
NOT TAKE STATES
NOT TAKE DIRECTION
TAKE DIRECTION

FIG. 9

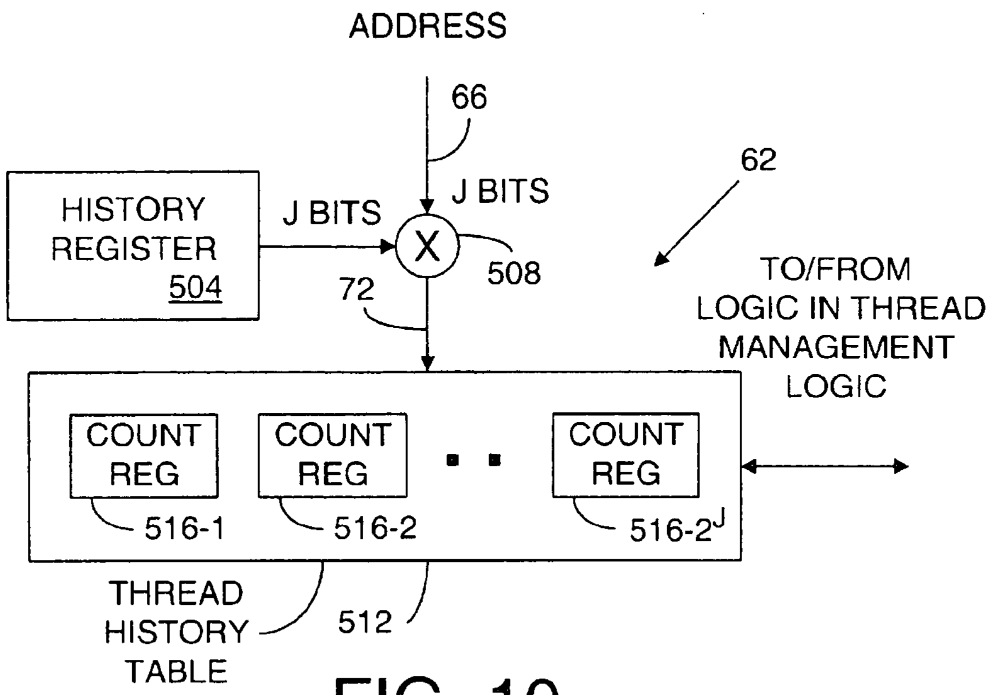

ADDRESS
66
J BITS
62
HISTORY REGISTER 504
J BITS
X
508
72
TO/FROM LOGIC IN THREAD MANAGEMENT LOGIC
COUNT REG
COUNT REG
COUNT REG
516-1
516-2
516-2^J
THREAD HISTORY TABLE
512

FIG. 10

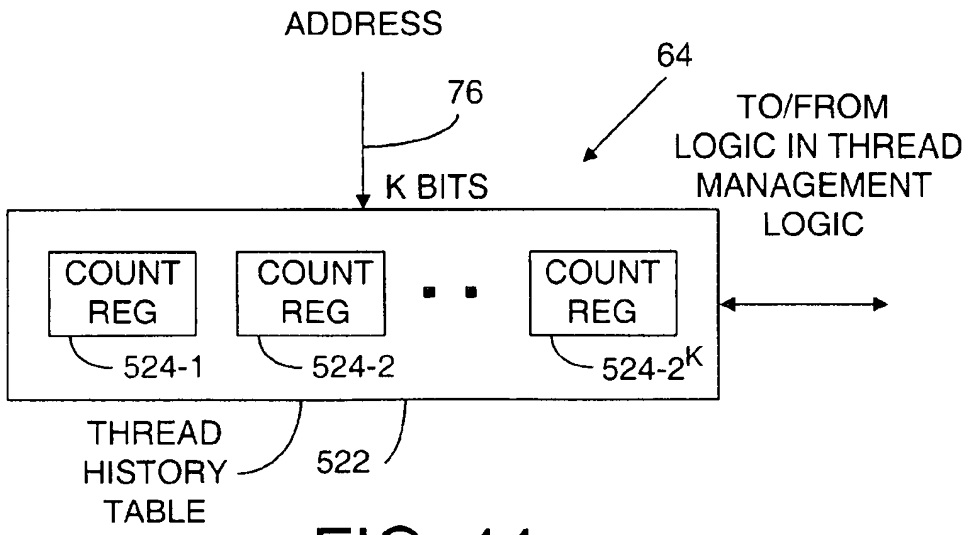

ADDRESS
64
76
TO/FROM LOGIC IN THREAD MANAGEMENT LOGIC
K BITS
COUNT REG
COUNT REG
COUNT REG
524-1
524-2
524-2^K
THREAD HISTORY TABLE
522

FIG. 11

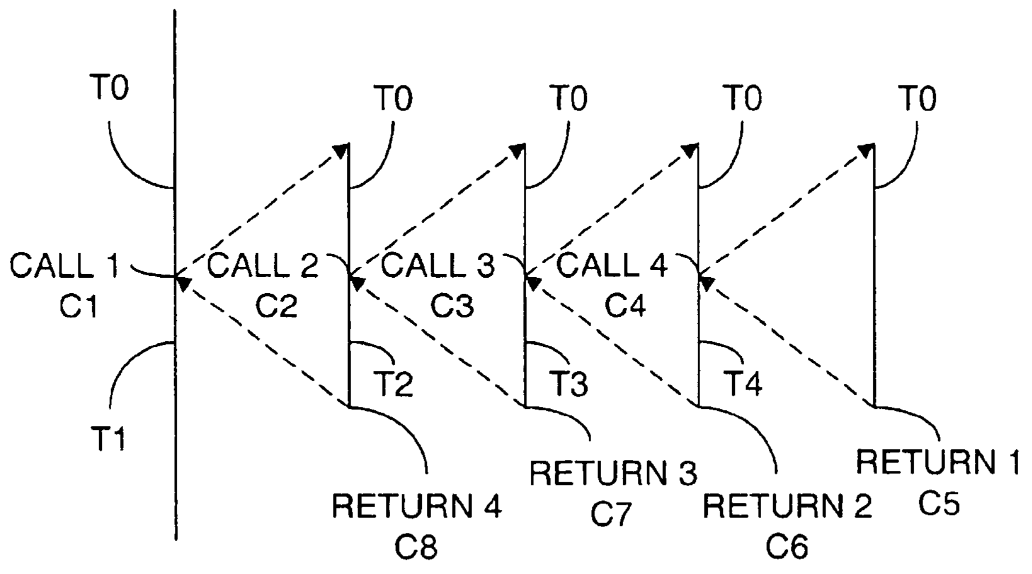

T0
T0
T0
T0
T0
CALL 1
C1
CALL 2
C2
CALL 3
C3
CALL 4
C4
T1
T2
T3
T4
RETURN 4
C8
RETURN 3
C7
RETURN 2
C6
RETURN 1
C5

FIG. 12

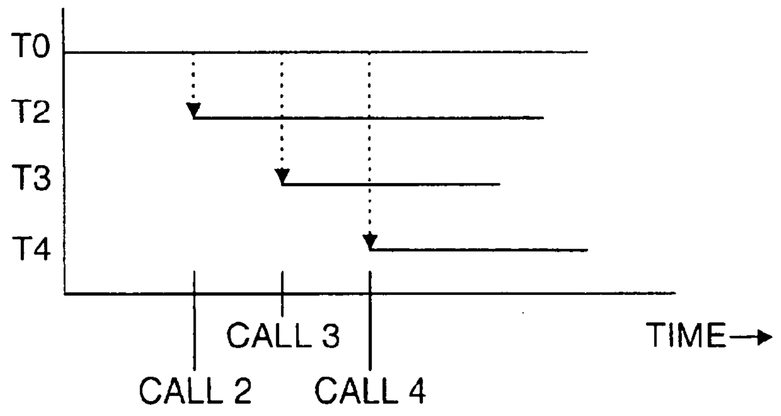

T0
T2
T3
T4
CALL 2
CALL 3
CALL 4
TIME

FIG. 13

144 TEMP REG

PC	THD	DIST CNT	SPWN CNT	JOIN CNT
	T0			

84 STACK

PC	THD	DIST CNT	SPWN CNT	JOIN CNT
	T1	1	C1	

FIG. 14A

144 TEMP REG

PC	THD	DIST CNT	SPWN CNT	JOIN CNT
	T0			

84 STACK

PC	THD	DIST CNT	SPWN CNT	JOIN CNT
	T2	1	C2	
	T1	2	C1	

FIG. 14B

144 TEMP REG

PC	THD	DIST CNT	SPWN CNT	JOIN CNT
	T0			

84 STACK

PC	THD	DIST CNT	SPWN CNT	JOIN CNT
	T3	1	C3	
	T2	2	C2	
	T1	3	C1	

FIG. 14C

144 TEMP REG

PC	THD	DIST CNT	SPWN CNT	JOIN CNT
	T0			

84 STACK

PC	THD	DIST CNT	SPWN CNT	JOIN CNT
	T4	1	C4	C5
	T3	2	C3	
	T2	3	C2	
	T1	4	C1	

FIG. 14D

144 TEMP REG

PC	THD	DIST CNT	SPWN CNT	JOIN CNT
	T4		C4	C5

84 STACK

PC	THD	DIST CNT	SPWN CNT	JOIN CNT
	T3	2	C3	C6
	T2	3	C2	
	T1	4	C1	

FIG. 14E

144 TEMP REG

PC	THD	DIST CNT	SPWN CNT	JOIN CNT
	T3		C3	C6

84 STACK

PC	THD	DIST CNT	SPWN CNT	JOIN CNT
	T2	3	C2	C7
	T1	4	C1	

FIG. 14F

MULTITHREADING PROCESSOR WITH THREAD PREDICTOR

RELATED APPLICATION

The present application is a continuation-in-part of U.S. application Ser. No. 08/992,375, filed Dec. 16, 1997, now pending.

BACKGROUND OF THE INVENTION

1. Technical Field of the Invention

The present invention relates to processors and, more particularly, to creation and management of threads in a processor.

2. Background Art

Current superscaler processors, such as a microprocessor, perform techniques such as branch prediction and out-of-order execution to enhance performance. Processors having out-of-order execution pipelines execute certain instructions in a different order than the order in which the instructions were fetched and decoded. Instructions may be executed out of order with respect to instructions for which there are not dependencies. Out-of-order execution increases processor performance by preventing execution units from being idle merely because of program instruction order. Instruction results are reordered after execution.

The task of handling data dependencies is simplified by restricting instruction decode to being in-order. The processors may then identify how data flows from one instruction to subsequent instructions through registers. To ensure program correctness, registers are renamed and instructions wait in reservation stations until their input operands are generated, at which time they are issued to the appropriate functional units for execution. The register renamer, reservation stations, and related mechanisms link instructions having dependencies together so that a dependent instruction is not executed before the instruction on which it depends. Accordingly, such processors are limited by in-order fetch and decode.

When the instruction from the instruction cache misses or a branch is mispredicted, the processors have either to wait until the instruction block is fetched from the higher level cache or memory, or until the mispredicted branch is resolved, and the execution of the false path is reset. The result of such behavior is that independent instructions before and after instruction cache misses and mispredicted branches cannot be executed in parallel, although it may be correct to do so.

Multithreading processors such as shared resource multithreading processors and on-chip multiprocessor (MP) processors have the capability to process and execute multiple threads concurrently. The threads that these processors process and execute are independent of each other. For example, the threads are either from completely independent programs or are from the same program but are specially compiled to create threads without dependencies between threads. However, these processors do not have the ability to concurrently execute different threads from the same program that may have dependencies. The usefulness of the multithreading processors is thereby limited.

Accordingly, there is a need for multithreading processors that have the ability to concurrently execute different threads from the same program where there may be dependencies among the threads.

SUMMARY OF THE INVENTION

In one embodiment, a processor includes thread management logic including a thread predictor having state

machines to indicate whether thread creation opportunities should be taken or not taken. The processor includes a predictor training mechanism to receive retired instructions and to identify potential threads from the retired instructions and to determine whether a potential thread of interest meets a test of thread goodness, and if the test is met, one of the state machines that is associated with the potential thread of interest is updated in a take direction, and if the test is not met, the state machine is updated in a not take direction.

The thread management logic may control creation of an actual thread and may further include reset logic to control whether the actual thread is reset and wherein if the actual thread is reset, one of the state machines associated with the actual thread is updated in a not take direction. The final retirement logic may control whether the actual thread is retired, and wherein if the actual thread is retired, the state machine associated with the actual thread is updated in a take direction.

The circuitry may be used in connection with a multithreading processor that detects speculation errors involving thread dependencies in execution of the actual threads and re-executes instructions associated with the speculation errors from trace buffers outside an execution pipeline.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention will be understood more fully from the detailed description given below and from the accompanying drawings of embodiments of the invention which, however, should not be taken to limit the invention to the specific embodiments described, but are for explanation and understanding only.

FIG. **1** is a block diagram of a processor according to one embodiment of the invention.

FIG. **2** is a flow diagram of an example of two threads.

FIG. **3** is a flow diagram of another example of two threads.

FIG. **4** is a flow diagram of an example of six threads.

FIG. **5** is a graph showing overlapping execution of the threads of FIG. **6**.

FIG. **6** is a block diagram illustrating individual trace buffers according to one embodiment of the invention.

FIG. **7** is a block diagram of a details of certain components of the thread management logic and final retirement logic of FIG. **1** according to one embodiment of the invention.

FIG. **8A**, **8B**, **8C**, and **8D** illustrates trees to organize thread relationships of FIG. **4**.

FIG. **9** illustrates states of a state machine.

FIG. **10** is a correlated thread predictor according to one embodiment of the invention.

FIG. **11** is a simple thread predictor according to one embodiment of the invention.

FIG. **12** illustrates nested procedures.

FIG. **13** is a graph illustrating execution of the actual threads corresponding to potential threads in FIG. **12**.

FIGS. **14A**, **14B**, **14C**, **14D**, **14E**, and **14E** are graphical representations of predictor stacks according to one embodiment of the invention.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

A. System Overview

B. Thread Predictor and Training Mechanism

1. When an Actual Thread is Retired or Reset Without Retirement

2. When a Potential Thread does or does not Meet a Test of Thread Goodness

C. Additional Information and Embodiments

Incorporation by Reference: Page 3, line 12 to page 37, line 30 and FIGS. 1–38 of U.S. application Ser. No. 08/992,375, filed Dec. 16, 1997, now pending, are incorporated into the present specification and drawings by reference. However, the present invention is not limited to use in connection with the various examples provided in U.S. application Ser. No. 08/992,375.

Reference in the specification to "one embodiment" or "an embodiment" means that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one embodiment of the invention. The appearances of the phrase "in one embodiment" in various places in the specification are not necessarily all referring to the same embodiment.

Referring to FIG. 1, a processor **50** includes an execution pipeline **108** in which instructions may be executed speculatively. Examples of the speculation include data speculation and dependency speculation. Any of a wide variety of speculations may be involved. Processor **50** includes mechanisms, including in trace buffer **114**, to detect speculation errors (misspeculations) and to recover from them. When a misspeculation is detected, the misspeculated instruction is provided to execution pipeline **108** from trace buffers **114** through conductors **120** and is replayed in execution pipeline **108**. If an instruction is "replayed," the instruction and all instructions dependent on the instruction are re-executed, although not necessarily simultaneously. If an instruction is "replayed in full," the instruction and all instructions following the instruction in program order are re-executed. The program order is the order the instructions would be executed in an in order processor. When it is determined that an instruction should be replayed, instructions which are directly or indirectly dependent on that instruction are also replayed. The number of re-executions of instructions can be controlled by controlling the events which trigger replays. In general, the term execute may include original execution and re-execution. Results of at least part of the instructions are provided to the trace buffers. Final retirement logic **134** finally retires instructions in trace buffers **114** after it is assured that the instructions were correctly executed either originally or in re-execution. The invention is not restricted to any particular execution pipeline. For example, although professor **50** is an out-of-order processor for intra-thread execution, the invention could be used with an in order processor for intra-thread execution. Execution pipeline **108** may be any of a wide variety execution pipelines and may be a section of a larger pipeline. Execution pipeline **108** may be used in connection with a wide variety of processors.

As examples, the following describes certain processors in which the prediction and training circuitry of the present invention may be included. However, the invention is not restricted to use in such processors.

A. System Overview

1. Creation of Threads and Overview of Pipeline

Instructions are provided through conductors **102** to an instruction cache (I-cache) **104**. A decoder **106** is illustrated as receiving instructions from I-cache **104**, but alternatively could decode instructions before they reach I-cache **104**. Depending on the context and implementation chosen, the term "instructions" may include macro-operations (macro-op), micro-operations (uops), or some other form of instructions. Any of a variety of instruction sets may be used including, but not limited to, reduced instruction set computing (RISC) or complex instruction set computing (CISC) instructions. Further, decoder **106** may decode CISC instructions to RISC instructions. Instructions from I-cache **104** are provided to pipeline **108** through MUX **110** and to trace buffers **114** through conductors **118**.

A trace is a set of instructions. A thread includes the trace and related signals such as register values and program counter values.

Thread management logic **124** creates different threads from a program or process in I-cache **104** by providing starting counts to program counters 112A, 112B, . . . , 112X, through conductors **130** (where X represents the number of program counters). As an example, X may be **4** or more or less. Thread management logic **124** also ends threads by stopping the associated program counter. Thread management logic **124** may cause the program counter to then begin another thread. Portions of different threads may be concurrently read from I-cache **104**.

To determine where in a program or process to create a thread, thread management logic **124** may read instructions from decoder **106** through conductors **128**. The threads may include instructions inserted by a programmer or compiler that expressly demarcate the beginning and ending of threads. Alternatively, thread management logic **124** may analyze instructions of the program or process to break up a program or process supplied to I-cache **104** into different threads. For example, branches, loops, backward branches, returns, jumps, procedure calls, and function calls may be good points to separate threads. Thread management logic **124** may consider the length of a potential thread, how many variables are involved, the number of variables that are common between successive threads, and other factors in considering where to start a thread. Thread management logic **124** may consider the program order in determining the boundaries of threads. The program order is the order the threads and the instructions within the threads would be executed on an in order processor. The instructions within the threads may be executed out of order (contrary to program order). The threads may be treated essentially independently by pipeline **108**. Thread management logic **124** may include a prediction mechanism including a history table to avoid making less than optimal choices. For example, thread management logic **124** may create a thread and then later determine that the thread was not actually part of the program order. In that case, if the same code is encountered again, the prediction mechanism could be used to determine whether to create that same thread again.

Dynamically creating threads is creating threads from a program that was not especially written or compiled for multithreading, wherein at least one of the threads is dependent on another of the threads. The program may originate from off a chip that includes execution pipeline **108** and thread management logic **124**. Dynamically creating the threads, executing the threads, and detecting and correcting speculation errors in the execution is referred to as dynamic multithreading.

Thread management logic **124** may use a combination of dynamic creation of threads and use of express instruction hints from a compiler or programmer. Thread management logic **124** would ordinarily create threads dynamically. However, the express instruction hints could be used occasionally. One way in which they may be used is in connection with if-then-else code in which a thread could be created beginning with the code following the conclusion of the else statement. It would take considerable complexity for thread management logic **124** to look forward to these instructions, but can be done in a compiler.

An advantage of using both a compiler and hardware for multithreading is that the compiler does not have to be conservative. If only the compiler does multithreading, the compiler must be conservative to handle worst cases. If the hardware assists the compiler, the compiler can be more aggressive because the hardware will detect and replay instructions associated with an error such as misspeculation.

FIG. 2 illustrates a thread T1 that includes a conditional backward branch instruction. In program order, thread T2 is executed following the conditional branch instruction when the branch is not taken. In time order, thread T2 is executed speculatively beginning at the time thread T1 first reaches the conditional branch instruction. Therefore, portions of thread T1 and T2 are executed concurrently. If thread T2 involves misspeculations, the effected instructions of thread T2 are replayed. Thread management logic 124 may monitor the count of the program counters through conductors 130. A purpose of monitoring the count is to determine when a thread should end. For example, when the condition of the conditional branch is not met, if the program counter of thread T1 were allowed to continue, it would advance to the first instruction of thread T2. Therefore, thread management logic 124 stops the program counter of thread T1 when the condition is not met.

FIG. 3 illustrates a thread T1 that includes a function call instruction. In program order, when the call instruction is reached, the program counter jumps to the location of the function and executes until a return instruction, at which time the program counter returns to the instruction after the call. In program order, thread T2 begins at the instruction following the return. In time order, thread T2 is executed speculatively beginning at the time thread T1 first reaches the call. If thread T2 involves misspeculations, the effected instructions of thread T2 are replayed. Thread T1 ends when its program counter reaches the first instruction of thread T2.

FIG. 4 illustrates threads T1, T2, T3, and T4 which are part of a section of a program. Different program counters produce threads T1, T2, T3, and T4. Thread T1 includes Instructions to point A (function call instruction) and then from point B, to point C (conditional backward branch instruction), to point D and to point C again (the loop may be repeated several times). Thread T2 begins at the instruction that in program order is immediately after the return instruction of the function that is called at point A. Thread T3 begins at the fall through instruction in memory following the conditional backward branch of point C and continues to point E, to point F, to point G, to point H, and to point I, which is a return instruction to the instruction immediately following point A where thread T2 begins. Thread T4 begins at the fall through instruction in memory following the conditional backward branch at point E. Thread T5 starts following a backward branch instruction at point J and thread T6 starts following a function call at point K. It is assumed that there are only four trace buffers.

As illustrated in FIG. 5, portions of threads T1, T2, T3, and T4 are fetched, decoded, and executed concurrently. The threads are fetched, decoded, and executed out of order because the program order is not followed. In time order, execution of threads T2, T3, and T4 begins immediately following instructions at points A, C, and E, respectively. The vertical dashed lines show a parent-child relationship. Threads T2, T3, and T4 are executed speculatively by relying on data in registers and/or memory locations before it is certain that the data is correct. Processor 50 has mechanisms to detect misspeculation and cause misspeculated instructions to be replayed. It turns out that thread T4 is not part of the program order. Thread T4 may be executed

until thread management logic 124 determines that thread T4 is not part of the program order. At that time, thread T4 may be reset without retirement under the control of reset logic 68 in thread management logic 124 and the resources that held or processed thread T4 in processor 50 may be deallocated and then allocated for another thread. In program order, threads T1, T2, and T3 would be executed as follows: first thread T1, then thread T3, and then thread T2.

Referring to FIG. 1, instructions from MUX 110 are received by rename/allocate unit 150 which provides a physical register identification (PRID) of the renamed physical register in register file 152. The PRID is provided to trace buffer 114 through bypass conductors 126. Allocation involves assigning registers to the instructions and assigning entries of the reservation stations of schedule/issue unit 156. Once the operands are ready for a particular instruction in the reservation stations, the instruction is issued to one of the execution units (e.g., integer, floating point) of execution units 158 or a memory execution pipeline which includes address generation unit (AGU) 172, memory order buffer (MOB) 178 (including load buffers 182 (one for each active thread) and store buffers 184 (one for each active thread)) and data cache 176. Depending on the instructions, operands may be provided from register file 152 through conductors 168. Under one embodiment of the invention, dependent instructions within a thread may be so linked that they are not executed out-of-order. However, dependent instructions from different threads may be concurrently fetched, decoded, and executed out-of-order.

For high performance, reservation stations and related mechanisms are designed to have both low latency and high bandwidth issue of instructions. The latency and bandwidth requirements place restrictions on the number of instructions that can be waiting in the reservation stations. By positioning trace buffers 114 outside pipeline 108, a large number of instructions can be available for execution/replay without significantly decreasing throughput of pipeline 108. The effect of latency between execution pipeline 108 and trace buffers 114 can be reduced through pipelining.

The result of an execution and related information are written back from the memory through MUX 192 and conductors 196 (in the case of loads) and from writeback unit 162 through conductors 122 (in the case of other instructions) to trace buffers 114. The results and related information may also be written to register file 152 and associated re-order buffer (ROB) 164. Once the result and information of an instruction are written to register file 152 and ROB 164, the instruction is retired in order as far as pipeline 108 is concerned. This retirement is called a first level or initial retirement. At or before the first level retirement, resources for the retired instruction in schedule/issue unit 156 including the reservation stations, register file 152, and ROB 164 are deallocated. However, all needed details regarding the instruction are maintained in trace buffers 114 and MOB 178 until a final retirement, described below.

A dependency exists between a later thread and an earlier thread when in program order, data used in the later thread is produced in the earlier thread. The data may have been produced in the earlier thread through a memory or non-memory instruction. For example, the later thread may be dependent on the earlier thread if a load instruction in the later thread has the same address as a store instruction in the earlier thread. The later thread may also be dependent on the earlier thread if an instruction in the later thread involves a register that was modified in the earlier thread. Likewise, a later instruction is dependent on an earlier instruction when in program order the later instruction uses data produced by the earlier instruction. The word "dependency" is also used in the phrase "dependency speculation." An example of a dependency speculation is speculating that there is no dependency between a load instruction and an earlier store instruction. Address matching is an example of a technique for checking for dependency speculation errors. An example of data speculation is speculating that the data in a register is the correct data. Register matching is an example of a technique for checking for data speculation errors.

Referring to FIG. 6, trace buffers **114** include trace buffers **114**A, **114**B, **114**C, . . . , **114**Y, where Y presents the number of trace buffers. For example, if Y=4(i.e., Y=D), there are 4 trace buffers. If Y is less than 3, trace buffers **114** does not include all the trace buffers shown in FIG. 6. Y may be the same as or different than X (the number of program counters). Trace buffers **114** may be a single memory divided into individual trace buffers, or physically separate trace buffers, or some combination of the two.

Referring to FIG. 6, trace buffers **114**A, **114**B, . . . , **114**Y receive instructions through conductors **118**A, **118**B, . . . **118**Y, which are connected to conductors **118**. There may be demultiplexing circuitry between conductors **118**A, **118**B, . . . , **118**Y and conductors **118**. Alternatively, enable signals may control which trace buffer is activated. Still alternatively, there may be enough parallel conductors to handle parallel transactions. Trace buffers **114**A, **114**B, . . . , **114**Y supply instructions and related information for replay to pipeline **108** through conductors **120**A, **120**B, . . . **120**Y, which are connected to conductors **120**. It is noted that multiple instructions from trace buffers **114** may concurrently pass through conductors **120** and MUX **110** for re-execution. At the same time, multiple instructions from decoder **106** may also pass through MUX **110** for the first time. A thread ID and instruction ID (instr ID) accompany each instruction through the pipeline. A replay count may also accompany the instruction. In the case of load and store instructions, a load buffer ID (LBID) and a store buffer ID (SBID) may also accompany the instruction. In one embodiment, the LBID and SBID accompany every instruction, although the LBID and SBID values may be meaningless in the case of instructions which are not loads or stores. As described below, a PRID or value may also accompany an instruction being re-executed.

Trace buffers **114**A, **114**B, . . . , **114**Y receive PRID, LBID, and SBID values from rename/allocate unit **150** through bypass conductors **126**A, **126**B, . . . **126**Y, which are connected to conductors **126**. Trace buffers **114**A, **114**B, . . . , **114**Y receive writeback result information and related signals through conductors **122**A, **122**B, . . . , **122**Y, which are connected to conductors **122**, and through conductors **196**A, **196**B, . . . , **196**Y, which are connected to conductors **196**. Replay signals are provided through conductors **194**A, **194**B, . . . , **194**Y, which are connected to conductors **194**. Multiplexing and/or enable circuitry and/or a substantial number of parallel conductors may be used in conductors **120**, **126**, **122**, **194**, and **196**. The trace buffers are not necessarily identical.

2. Misspeculation Detection Circuitry

The following provides examples of misspeculation detection circuitry, but the invention is not limited to use with such detection circuitry. Trace buffers **114** include detection circuitry to detect certain speculation errors. According to one embodiment of the invention, each trace buffer has an output register file that holds the register context of the associated thread and an input register file to receive the register context of the parent thread output

register. The register context is the contents or state of the logical registers. The contents of the output register file is updated often, perhaps each time there is a change in a register. The contents of the input register file is initialized when a thread is created and updated only after a comparison, described below.

An output register file and input register file may include a Value or PRID field and a status field. The status field indicates whether a valid value or a valid PRID is held in the Value or PRID field. A comparator compares the contents of input register file (in trace buffer **114**B) for a current thread with the contents of output registers for an immediately preceding thread in program order. The comparison can be made at the end of the execution of the immediately preceding thread or during the execution of the preceding thread. The comparison is also made at the end of the retirement of the preceding thread. In one embodiment, the comparison is only made at the end of the retirement of the preceding thread. In another embodiment, the comparison is also made at intermediate times. In other embodiments, the comparison is made continuously during final retirement of the preceding thread.

Various events could trigger a comparison by the comparator. The comparison is made to detect speculation errors. If there is a difference between the speculative input register 9 file and the output register file from the preceding thread, values of one or more output registers of the immediately preceding thread have changed and a replay is triggered. Trace buffer **114**B causes the effected instructions to be replayed with the changed register values. There is no guarantee the changed values are the ultimately correct values (i.e., the values that would have been produced in an in order processor). The instructions may need to be replayed again, perhaps several times.

When replay triggering logic determines that a source operand (or other input value) has been mispredicted, it triggers the corresponding trace buffer (such as trace buffer **114**B) to dispatch those instructions that are directly or indirectly dependent on the mispredicted source operand to be replayed in pipeline **108**.

In one embodiment, an algorithm that identifies dependent instructions in a way that is similar to register renaming. Instead of a mapping table, a dependency table is used. The table contains a flag for each logical register that indicates if the register depends on the mispredicted value. At the start of a recovery sequence, the flag corresponding to each mispredicted register is set. The dependency table is checked as instructions are read from the trace buffer. If one of the source operand flags is set, an instruction depends on the mispredicted value. The instruction is selected for recovery dispatch and its destination register flag is set in the table. Otherwise, the destination register flag is cleared. Only the table output for the most significant instruction within the block can be relied upon all the time. Subsequent instructions may depend on instructions ahead of them within the block. A bypass at the table output is needed to handle internal block dependencies. If the dependency table reaches a state during the recovery sequence in which all the dependency flags are clear, the recovery sequence is terminated.

The identified instructions are dispatched from the trace buffer for execution in the order the instructions exist in the trace buffer (which is the program order). However, the instructions may be executed out of order under the control of schedule/issue unit **156**, as in any out-of-order processor. Control bits are appended to the instruction dispatched from the trace buffer to indicate to rename/allocate unit **150** whether to (1) do register renaming, (2) bypass the rename alias table lookup in rename/allocate unit **150** and instead use the PRID from the corresponding trace buffer, or (3) bypass renaming completely and use the value from the trace buffer as if it where a constant operand in the instruction.

3. Information Regarding Thread Management Logic and Final Retirement

The following are some details regarding thread management logic and final retirement that may be used in connection with some embodiment of the invention. However, the invention is not limited to such details. In one embodiment, thread management logic **124** uses a tree structure (such as tree structure **58** in FIG. 7) to keep track of thread order. Under the tree structure, the program order (which is also the retirement order) flows from top to bottom, and a node on the right is earlier in program order than a node on the left. A root is the first in program order. A tree is an abstract idea, whereas a tree structure is circuitry that implements the tree.

For example, FIGS. **8**A, **8**B, **8**C, and **8**D illustrate trees structures for threads of FIG. **4** at different times. FIG. **8**A illustrates the tree structure at time t1. Thread T2 is added to the tree before thread T3 is added to the tree. Thread T4 is added to the tree after thread T3 is added to the tree. Threads T2 and T3 are children of thread T1. Thread T4 is a child of thread T3. Following the rules of top to bottom and right to left, the program and retirement orders are thread T1, T3, T4, and T2. FIG. **8**B illustrates the tree structure at time t2 assuming that thread T4 is reset before thread T1 retires. The program and retirement orders are thread T1, T3, T2, and T5. FIG. **8**C illustrates the tree structure at time t2 assuming that thread T1 retires before thread T4 is reset. The program and retirement orders are thread T3, T4, T2, and T5. FIG. **8**D illustrates the tree structure at time t3, which is after the time thread T1 retires and thread T4 is reset. The program. and retirement orders are T3, T2, T5 and T6.

Threads begin at the instruction following a backward branch or a function call. That is, threads begin at the next instruction assuming the backward branch were not taken or the function was not called (as illustrated by threads T2 in FIGS. **2** and **3**). In so doing, from the perspective of a thread (node), the program order of children nodes of the thread are in the reverse of the order in which the threads were started (created).

In one embodiment, three events may cause a thread to be removed from the tree: (1) A thread at the root of the tree is removed when the thread is retired. When the thread at the root is retired, the thread (node) that is next in program order becomes the root and nodes are reassigned accordingly. (2) A thread that is last in program order is removed from the tree to make room for a thread higher in program order to be added to the tree. In this respect, the tree acts as a last-in-first-out (LIFO) stack. (3) A thread may be reset and thereby removed from the tree when it is discovered that the program counter of its parent thread is outside a range between a start count and an end count.

An instruction is finally retired from trace buffer **114** when all the instructions for all previous threads have retired and all replay events that belong to the instruction have been serviced. Stated another way, an instruction is finally retired when it can be assured that the instruction has been executed with the correct source operand. Threads are retired in order. For example, an instruction in thread X cannot be retired until all the previous threads have been retired (i.e., the instructions of all the previous threads have been retired). The instructions within a thread are retired in order, although instructions that are all ready for retirement may be retired simultaneously.

Final retirement is controlled by final retirement logic **134**. In one embodiment of the invention, final retirement includes (1) commitment of results to in order register file, (2) service interrupts, exceptions, and/or branch mispredictions; (3) deallocation of trace buffer and MOB **178** resource entries; and (4) signaling the MOB to mark stores as retired and to issue then to memory. Deallocating entries may involve moving a head pointer. Store instructions in MOB **178** are not deallocated until after it is certain that associated data is copied to data cache **176** or other memory.

4. Memory Order Buffer

The invention is not limited to any particular type of memory order buffer (MOB). However, to the extent load and store instructions are executed speculatively, there should be a mechanism to detect misspeculations and to allow for replay of effected instructions. For example, there may be a mechanism to detect when a load is made from a memory location before a store to the same location (which occurred earlier in program order but later in time order). For example, MOB **178** includes a MOB for each thread, where each MOB includes a load buffer arid store buffer to hold copies of load and store instructions of the traces in trace buffers **114**A, **114**B, . . . , **114**Y, respectively. Replay conductors **194** provide signals from MOB **178** to trace buffers **114** alerting trace buffers **114** that a load instruction should be replayed. To ensure ultimate correctness of execution, MOB **178** includes mechanisms to ensure memory data coherency between threads.

B. Thread Predictor and Training Mechanism

Referring to FIGS. **1** and **7**, processor **50** includes thread predictor **54** to make predictions regarding whether a thread should be created in response to reception of a thread spawning opportunity instruction. Examples of spawning opportunity instructions include function calls, backward branches for loops, and compiler inserted instructions suggesting or mandating spawning of a thread. (Spawning a thread means creating a thread.) Thread predictor **54** is used by thread management logic to increase the likelihood that executed threads will retire. The invention also includes a training mechanism **56** to train thread predictor **54** to help decrease the probability that thread management logic **124** will create threads that have relatively few instructions or relatively little overlap with the spawning thread (the thread with the spawning opportunity instruction). These predictions help increase processor performance.

The following terminology is used with respect to one embodiment of the invention. An actual thread is a thread that is created through a program counter and executed at least in part. An actual thread is either retired or reset without retirement. A thread creation opportunity is an opportunity to create an actual thread. Thread management logic **124** decides whether or not to take the thread creation opportunity and create an actual thread in response to detecting a spawning opportunity instruction from decoder **106** (or from other circuitry). A potential thread is a group of retired instructions analyzed in a predictor training mechanism. The potential thread is associated with a retired spawning opportunity instruction. The set of instructions included in a potential thread may or may not correspond exactly to a set of instructions in a recently retired actual thread. Although the specification does not always explicitly identify a thread as being actual or potential, it is clear from the context.

Thread predictor **54** includes numerous state machines corresponding to different spawning opportunity instructions. (In some embodiments, more than one state machine may correspond to the same spawning opportunity instruction, such with the correlated predictor described below.) The state machines may share some circuitry in common. FIG. 9 illustrates states 0, 1, 2, . . . , 7 of state machine. As an example, states 0–3 may be not take states and states 4–7 may be take states (although the number of take and not take states do not have to be equal). A take state means a thread creation opportunity is to be taken and a not take state means the thread creation opportunity is not to be taken. When the state machine is updated in the take direction, the state changes from a lower to a higher state, unless the state is already at state 7, in which case it remains at state 7. When the state machine is updated in the not take direction, the state changes to from a higher to a lower state, unless the state is already at state 0, in which case it remains at state 0. Examples of updating in the take direction include state 0 to state 1, state 3 to state 4, and state 7 to State 7. Examples of updating in the not take direction includes state 6 to state 5, state 4 to state 3, and state 0 to state 0. As an example, if the state machine included three bit saturated counters, states 0–7 might be binary states "000," "001," "010," "011," "100," "101," "110," and "111," respectively

A state machine are updated in response to two events (1) an actual thread is retired or reset without retirement or (2) a potential thread does or does not meet a test of thread goodness. If the state machine is a counter, updating may include increasing or decreasing a count. However, updating a state machine does not always result in a change of the state of the state machine (e.g., the state in a saturating counter state does not go above or below certain counts).

1. When an Actual Thread is Retired or Reset Without Retirement

When an actual thread is retired, that is evidence that the thread would retire again if it were created again in response to the same spawning opportunity instruction, particularly if it has the same thread lineage. The thread lineage of a thread of interest includes a parent thread and perhaps a grandparent thread. The parent thread is the thread containing the spawning opportunity instruction. The grandparent thread is the parent of the parent. The thread lineage is represented as address bits of the thread of interest, parent, and perhaps grandparent. However, not all the bits need be used (e.g., only part of the patent and grandparent bits might be used) and the bits that are used may be acted on by some function to help avoid aliasing with respect to the thread predictor.

In response to an actual thread retiring, a signal is provided to thread predictor 54 to update the state machine (e.g., incrementing the state of the state machine) to indicate a successfully taken thread creation opportunity. However, in general, there is no guarantee the thread would retire again if it were executed again. If a thread is reset, that is evidence that a thread would be reset again if it were created again, particularly if it has the same lineage. Accordingly, in response to an actual thread being reset, a signal is provided to thread predictor 54 to update the state machine (e.g., decrementing the state) to indicated a unsuccessfully taken thread creation opportunity. However, in general, there is no guarantee the thread would be reset if executed again.

Referring to FIG. 7, in one embodiment, thread predictor 54 includes a correlated predictor 62 and a simple predictor 64 (although in other embodiments only a correlated predictor or only a simple predictor might be used). The following provides details of specific examples of correlated and simple predictors. However, the invention may be implemented with different details.

Correlated predictor 62 and/or simple predictor 64 are updated in response to an actual thread retiring or resetting and are updated in response to a potential thread meeting or not meeting a test of thread goodness. Correlated predictor 62 and/or simple predictor 64 are read in response to a thread creation opportunity being detected. Referring to FIG. 10, a correlated predictor 62 includes a thread history table 512 including 2^j count registers 516-1, 516-2, . . . 516-2^j, which may be part of finite state machines (e.g., up/down saturating counters). When a thread is retired or reset, a signal is provided on conductors 142 (as shown in FIG. 7) to update the state of correlated predictor 62 by updating the count in the count register that corresponds to the actual thread that is retired or reset. A read-modify-write approach may be used to update the value of the appropriate count register. Logic in thread management logic 124 controls updating and reading of states. When a spawning opportunity instruction is detected, thread management logic 124 reads the corresponding count register to determine whether or not to take the thread creation opportunity based at least in part on the state of correlated predictor. For example, if the count in the count register has one relationship (e.g.,>or≧) to a threshold, correlated predictor 62 is in a "take state" and the actual thread is created. If the count has another relationship (e.g.,<or≦) to the threshold, correlated predictor 62 is in a "not take state" and an actual thread is not created. In one embodiment, even if the state is a "take state," thread management logic 124 will not cause a thread to be created if it means resetting a thread earlier in program order. In another embodiment, the thread will be created even if it means resetting a thread earlier in program order.

In one embodiment, a particular one of the count registers 516 of thread history table 512 is indexed through a signal on conductors 72, which is the result of an exclusive-OR (XOR) function 508. One input to XOR function 508 is the J least significant bits (LSBs) of an address on conductors 66 representing an actual thread (for updating a count register following retirement or reset), a potential thread (for updating a count register following a determination of whether a potential thread meets the test), or thread creation opportunity (for reading a count register), depending on the situation. The address provided to XOR function 508 does not necessarily have to be the first address of an actual thread, retired thread, or thread creation opportunity.

Another input is a thread lineage from history register 504. The bits in history register 504 may include some function of bits of the parent and grandparent of actual thread, potential thread, or thread creation opportunity. For example, the function may be the X LSBs from the first address of the parent and fewer LSBs from the first address of the grandparent thread of the thread creation opportunity, either concatenated or combined through, for example, an XOR function. In the case of a retired or reset actual thread, the thread lineage value for history register 504 may come from the tree structure. Each time a new thread is added to the tree structure, a thread lineage field associated with the tree structure can hold the thread lineage value associated with the thread. Then when the thread leaves the tree structure, the thread lineage value can be written into history register 504. In the case of a thread creation opportunity, the thread lineage value to be written into history register 504 can be determined from the address of the thread creation opportunity and the parent and perhaps grandparent in the tree structure. In the case of a potential thread, the thread lineage value to be written into history register 504 can be copied from a stack described below in connection with potential threads.

XOR function 508 is used to reduce the chances of aliasing in which the LSBs of the addresses of two thread creation opportunities and their parent and grandparent threads have similar values. An XOR function 508 does not have to be used. For example, the address of the thread creation opportunity could index a row and history register **504** could index a column of count registers. Various arrangements may be used.

A reason for using bits from the parent and grandparent threads is that there may be a correlation between the lineage of a thread creation opportunity and whether the thread creation opportunity will be retired if it is spawned.

Referring to FIG. 11, as an example, simple predictor **64** includes a thread history table **522** which includes 2^K count registers. (K may be equal to or different than J.) An address representing the actual thread, potential thread, or thread opportunity is received on conductors **76**. The count registers are updated and read as described above in connection with the correlated predictor. Initially, thread management logic **124** might only consider the counts of simple predictor **64**. Logic in thread management logic **124** controls updating and reading of simple predictor **64** and decides whether to create a thread based at least in part on the state of simple predictor **64** (e.g., the counts in the appropriate count registers). Then, as counts of correlated predictor **62** become meaningful, the counts of simple predictor **64** might be ignored.

2. When a Potential Thread does or does not Meet a Test of Thread Goodness

When an actual thread is executed, it will be known whether the thread is retired or reset without retirement. However, when a thread creation opportunity is not taken, the retirement or resetting of the non-created thread cannot be observed. Nonetheless, predictor training mechanism **56** analyzes retired instructions to make a prediction of whether it would be a good use of processor resources to create an actual thread from a potential thread.

Referring to FIG. 7, in one embodiment, final retirement logic **134** includes predictor training mechanism **56** which provides signals indicating whether potential threads meet a test of thread goodness. If the test of thread goodness is met, correlated predictor **62** and/or simple predictor **64** of thread predictor **54** are updated in a take direction (e.g., the count is incremented unless it is already a maximum). If the test is not met, correlated predictor **62** and/or simple predictor **64** are updated in a not take direction (e.g., the count is decremented unless it is a minimum). Referring to FIGS. **10** and **11**, in one embodiment, the address of a thread spawning opportunity instruction of the potential thread is used to index thread history tables **512** and **522**. In the case of correlated predictor **62** in FIG. **11**, the potential thread address is applied to an input to XOR function **508**. History register **504** holds bits related to the lineage of the potential threads.

In one embodiment, a potential thread is not analyzed if it's spawning opportunity instruction is the spawning opportunity instruction of an actual thread that is retiring or has just retired. In another embodiment, the potential thread is analyzed even if its spawning opportunity instruction is the spawning opportunity instruction of an actual thread that is retiring or has recently retired. In such a case, thread predictor **54** might only be updated in response to either (1) retirement or resetting of the actual thread or (2) the potential thread meeting or not meeting the test of thread goodness, but not (1) and (2). In another embodiment, thread predictor **54** might be updated twice in response to both (1) and (2).

Predictor training mechanism **56** includes a retired thread monitor **82** to analyze retired instructions and to identify potential threads in the retired instructions in the same way that thread management logic **124** identifies thread creation

opportunities in instructions. Predictor training mechanism **56** determines whether the potential threads meet a test of thread goodness.

In one embodiment, the test of thread goodness is met if certain criteria of thread goodness are met. Examples of such criteria include criteria (1), (2), and (3), described below. In another embodiment, there is only one criterion of thread goodness (e.g., criterion (1)) and the test is met if the criterion is met. In yet another embodiment, there are two criteria (which may include one or more of criteria (1), (2), and (3)), and the test is met if the two criteria are met. There may be more than three criteria. In still another embodiment, at least one but fewer than all of multiple criteria must be met for the test to be met. The test may change depending on circumstances.

FIG. **12** illustrates an example with which criteria (1), (2), and (3) may be described. FIG. **12** illustrates potential threads T0, T1, T2, T3, and T4 which include nested functions or procedures initiated by calls call 1, call 2, call 3, and call 4. It is assumed there are only four trace buffers **114A**, **114B**, **114C**, and **114D** in processor **50**. The program and retirement order is T0, T4, T3, and T1. Without the predictor of the invention, the time order would be T0, T1, T2, T3 and then T1 would be reset to make room for T4. However, with the invention, it can be predicted that T1 would be reset (because it fails criterion (1) discussed below) so that if each of threads T0–T4 is actually created and the T1 thread creation opportunity instruction is predicted "not taken," in thread predictor **54**, the time order would be T0, T2, T3, and T4. Referring to FIG. **13**, in the example, thread T1 would not be created, but rather would be included as part thread T2.

A retired instruction counter **86** is incremented as instructions are retired. Table 1, below, lists the spawning opportunity instruction and joining opportunity instruction of threads T1–T4 and the count of instruction counter 86 at the instruction, where C1<C2<C3. . .<C8. In one embodiment, the joining opportunity instruction is the instruction of the previous potential thread in program order that joins with the potential thread of interest. For example, return 1 is the joining instruction of potential thread T4.

TABLE 1

Thread	Spawning opportunity instruction	Count for spawning opportunity instruction	Joining opportunity instruction	Count for joining opportunity instruction
T1	call 1	C1	Return 4	C8
T2	call 2	C2	Return 3	C7
T3	call 3	C3	Return 2	C6
T4	call 4	C4	Return 1	C5

Criterion (1) Number of potential threads between spawn and join (stack depth) Criterion (1) concerns whether it would be likely that a potential thread, which is actually spawned, would be reset before retiring to make room for a thread earlier in program order. If so, the potential thread does not meet the criterion (1). Referring to FIGS. 7 and **14A–14F**, a stack 84 may be used to keep track of a potential thread distance count. (Stack **84** is illustrated with only four entries., but may have more.) In one embodiment, stack **84** includes fields for a program counter (PC) value of the first address of the potential thread of interest; an indicia of the thread (e.g., T1); a distance count (dist cnt) which represents the number of intervening potential threads (note that the value changes as additional spawning opportunity instructions are received); a spawn count, which is the count of retired instruction counter **86** associated with the spawning opportunity instruction; and a join count, which is the count of retired instruction counter **86** associated with a joining opportunity instruction. Stack **84** may include others fields such as whether the spawning opportunity function is a call, backward branch, or special compiler instruction. A temporary register **144** holds the information regarding the potential thread most recently popped off stack **84**. The thread lineage can be determined from the PCs of the potential thread of interest and its parent and perhaps grandparent or stack **84** may include a field (as part of, in addition to, or in place of the PC) to hold the thread lineage.

Referring to FIGS. **12** and **14**A, as the call 1 instruction is detected, information (e.g., PC value, thread indicia, and spawn count) regarding thread T1 is pushed onto stack **84**. A "1" is placed in a distance counter (dist cnt) field. Referring to FIG. **14**B, as the call 2 instruction is detected, information regarding thread T2 is pushed onto the stack and the information regarding thread T1 is pushed deeper into stack **84**. A "1" is placed in the distance count field of thread T2 and the distance count value of thread T1 is incremented to a "2". Referring to FIGS. **14**C and **14**D, as the call 3 and call 4 instructions are detected, information regarding threads T3 and T4 are pushed onto the stack in order, and the information regarding thread T1 and T2 are pushed deeper. The distance count fields are incremented as new spawning opportunity instructions are received.

When a potential thread is joined by a thread earlier in program order, the information regarding the potential thread is popped off the stack. As threads are popped off the stack, the distance counts do not decrement, except as described below. Referring to FIG. **14**E, after the return 1 instruction is received, thread T0 joins thread T4 and the information regarding thread T4 is popped off stack **84**. The distance count of thread T4 is "1" which is less than the threshold of 4, so T4 meets criterion (1). Referring to FIG. **14**F, after the return 2 instruction is received, thread T4 joins thread T3 and the information regarding thread T3 is popped off stack 84. The distance count of thread T3 is "2" which is less than the threshold of 4, so thread T4 meets criterion (1).

Note, however, the distance count of thread T1 is 4, which is not less than the threshold of 4, so that thread T1 does not meet criterion (1) of thread goodness. In essence, this is because each of the four trace buffers are likely to be used by threads T0, T2, T3, and T4 so that if thread T1 were created, it would likely be reset to make room for one of threads T2, T3, or T4, which would result in a performance penalty.

Criterion (2) Number of instructions between spawn to join (amount of overlapping execution). If the number of instructions between the spawning opportunity instruction of the potential thread of interest and the joining instruction of the spawning potential thread (which contains the spawning opportunity instruction) to potential thread of interest is relatively small, there will not be much concurrent execution of the spawning thread and the potential thread of interest. In such a case, there will not be much performance benefit and it probably will not be worth spawning the potential thread of interest. Rather, when it is executed, the instructions of the potential thread of interest could be part of the spawning potential thread when executed as an actual thread. For example, referring to FIG. 12, if the number of instructions between call 4, and the first instruction of thread T4 is small, it probably would not be a good use of resources to spawn thread T4. Rather, thread T4 could be part of thread T0.

In one embodiment, the amount of overlap (or concurrent execution) of actual threads is estimated by the difference of the join count and the spawn count of a potential thread. These values are provided in fields of stack **84** and the difference is shown in Table 2, below:

TABLE 2

Potential Thread of Interest	Overlapping Execution of Spawning Thread and Potential Thread of Interest (Count of joining opportunity instruction minus count of spawning opportunity instruction	Size of Thread (Count of next joining opportunity instruction minus count of joining opportunity instruction of potential thread of interest)
T1	C8 - C1	Not known in the example
T2	C7 - C2	C8 - C7
T3	C6 - C3	C7 - C6
T4	C5 - C4	C6 - C5

In one embodiment, the difference is compared to a threshold value to determine if criterion (2) of thread goodness is met. Criterion (2) is not met if the difference has a first relationship (e.g., <or≦) to the threshold value and is met if the difference has a second relationship (>or≧) to the threshold value. One factor in determining the magnitude of the threshold value is the number of cycles taken to create and to retire or reset a thread.

Criterion (3) Number of instructions from join to next join (size of potential thread of interest). If the number of instructions of a potential thread of interest is relatively small, it is probably not worth spawning it as an actual thread. The size of the potential thread is compared to a threshold value to determine if criterion (3) of thread goodness is met. Criterion (3) is not met if the size has a first relationship (e.g.,<or≦) to the threshold value and is met if the size has a second relationship (>or≧) to the threshold value. One factor in determining the magnitude of the threshold value is the number of cycles taken to create and to retire or reset a thread. Note that the size is only an estimate of the size of an actual thread.

In one embodiment, the size of the potential thread of interest is the difference of the join count of the next thread in program order and the join count of the potential thread of interest. As mentioned, the join count is the count of retired instruction counter **86** at the joining opportunity instruction. The temporary register **144** is used because the join count of the later potential thread is not known until after the potential thread of the interest is popped off stack **84** and placed in temporary register **144**. For example, in FIG. **14**E, information regarding thread T4 is held in temporary register **144**. The size of thread T4 is C6–C5, where C6 is in stack **84** and C5 is in the temporary register. Values for the thread distance are in Table 2, above.

Referring to FIGS. **14**D and **14**E, as illustrated, the distance count values do not decrement when the thread at the top of the stack is joined. However, in one embodiment, if the thread does not meet criteria (2) and (3), the distance count values of the potential threads are decremented.

In a more sophisticated and complicated mechanism, for criteria (2) and (3), retired thread monitor **82** could consider the type of instructions as well as the number of instructions.

In one embodiment, when a state machine is updated in response to retirement or resetting of an actual thread, the state is changed by two states (e.g., state 3 to state 5, state 6 to state 4, state 0 to state 0), but when the state machine is updated in response to a potential thread meeting or not meeting the test, it is only changed by one state (e.g., state 3 to state 4, state 7 to state 7).

Although the example of FIG. 12 concerns functions, the criteria of thread goodness may also be determined for threads having a backward branch spawning opportunity instruction. For calls, a join may be determined by observing the instruction statically following the spawning opportunity instruction. For loops, it may be determined by observing the loops exit (where the current retired instruction is between the bounds of the start and end of the loop and the next retired instruction is outside these bounds and the next retired instruction is not a call).

C. Additional Information and Embodiments

Referring to FIG. 1, in one embodiment, processor **50** may include a branch prediction unit **60** as part of a fetch unit that includes decoder **106** and program counters (PCs) **112A**, **11B**, . . . , **112X**. Branch prediction unit **60** may include a branch predictor having a correlated predictor and/or a simple predictor. If a branch is correctly predicted taken, the predictor is incremented and if a branch is incorrectly predicted taken, the corresponding count register is decremented. The correlated predictor may contain a branch history shift register that is shifted with each branch.

In one embodiment, there is a return address stack for each thread, to provide a return address from functions. More than one return address stack may be used because in a processor that executes threads out of order, the return from a function in time order may occur before an earlier call in the program. Of course in program order, the call statement comes first. The contents of the return address stack of the spawning thread at the time of the spawn is copied into the return address stack of the spawned thread (i.e., the thread created in response to the spawning opportunity instruction in the spawning thread). Following the copy, returned addresses are pushed and popped on and off of the return address stacks of the spawned and spawning threads independently.

The circuits and details that are described and illustrated are only exemplary. Various other circuits and details could be used in their place. Further, there may be various design tradeoffs in size, latency, etc. There may be intermediate structure (such as a buffer) or signals between two illustrated structures. Some conductors may not be continuous as illustrated, but rather be broken up by intermediate structure. The borders of the boxes in the figures are for illustrative purposes. An actual device would not have to include components with such defined boundaries. For example, a portion of the final retirement logic could be included in the boxes labelled trace buffers. The relative size of the illustrated components is not to suggest actual relative sizes. Arrows show certain data flow in certain embodiments, but not every signal, such as data requests. Where a logical high signal is described above, it could be replaced by a logical low signal and vice versa.

The instructions within a thread may pass from decoder **106** to trace buffers **114** and execution unit **108** entirely in program order or in something other than program order.

The terms "connected," "coupled," and related terms are not limited to a direct connection or a direct coupling, but may include indirect connection or indirect coupling. The term "responsive" and related terms mean that one signal or event is influenced to some extent by another signal or event, but not necessarily completely or directly. If the specification states a component "may", "could", or "might" to be included, that particular component is not required to be included.

Those skilled in the art having the benefit of this disclosure will appreciate that many other variations from the foregoing description and drawings may be made within the

scope of the present invention. Accordingly, it is the following claims including any amendments thereto that define the scope of the invention.

What is claimed is:

1. A processor comprising:

thread management logic including a thread predictor having state machines to indicate whether thread creation opportunities should be taken or not taken; and

a predictor training mechanism to receive retired instructions and to identify potential threads from the retired instructions and to determine whether a potential thread of interest meets a test of thread goodness, and if the test is met, one of the state machines that is associated with the potential thread of interest is updated in a take direction, and if the test is not met, the state machine is updated in a not take direction.

2. The processor of claim **1**, wherein the predictor training mechanism determines whether the potential thread of interest meets one criterion of thread goodness and the test is met if the criterion of thread goodness is met.

3. The processor of claim **1**, wherein the predictor training mechanism determines whether the potential thread of interest meets at least one criterion of thread goodness of certain criteria of thread goodness, and the test is met if the at least one criterion of thread goodness is met.

4. The processor of claim **1**, wherein the predictor training mechanism determines whether the potential thread of interest meets certain criteria of thread goodness and the test is met if the criteria of thread goodness are met.

5. The processor of claim **1**, wherein the test involves determining whether a criterion of thread goodness is met, which criterion involves making an estimate as to whether the potential thread of interest would be reset to make room for another thread earlier in program order if the potential thread of interest were created as an actual thread.

6. The processor of claim **1**, wherein the test involves determining whether a criterion of thread goodness is met, which criterion involves determining a number of retired instructions between a spawning opportunity instruction and a joining opportunity instruction of the potential thread of interest, and comparing the number is to a threshold value.

7. The processor of claim **1**, wherein the test involves determining whether a criterion of thread goodness is met, which criterion involves determining a number of instructions in the potential thread of interest and comparing the number to a threshold value.

8. The processor of claim **1**, wherein the predictor training mechanism includes a stack onto which information representing potential threads is pushed when a retired spawn opportunity instruction is detected and from which information is popped off when a retired joining opportunity instruction is detected and the stack includes a distance count related to a number of potential threads meeting a criterion of thread goodness between a retired spawning opportunity instruction and joining opportunity instruction of the potential thread of interest.

9. The processor of claim **8**, wherein the predictor training mechanism includes a retired instruction counter that provides a count presenting the retired instructions and wherein the stack further includes fields holding count values for spawning opportunity instructions and joining opportunity instructions.

10. A processor comprising:

thread management logic to control creation of an actual thread, the thread management logic including reset logic to control whether the actual thread is reset and a thread predictor having state machines to indicate whether thread creation opportunities should be taken or not taken, and wherein if the actual thread is reset, one of the state machines associated with the actual thread is updated in a not take direction; and

final retirement logic to control whether the actual thread is retired, and wherein if the actual thread is retired, the state machine associated with the actual thread is updated in a take direction, the final retirement logic including a predictor training mechanism to receive retired instructions and to identify potential threads from the retired instructions and to determine whether a potential thread of interest meets a test of thread goodness, and if the test is met, one of the state machines that is associated with the potential thread of interest is updated in a take direction, and if the test is not met, the state machine associated with the potential thread of interest is updated in a not take direction.

11. The processor of claim **10**, wherein the state machine associated with the actual thread is the same state machine associated with the potential thread of interest if the actual thread and potential thread of interest have the same index into the state machine.

12. The processor of claim **11**, wherein the index of the actual thread includes bits of an address of the actual thread and the index of the potential thread includes bits of an address of the potential thread.

13. The processor of claim **12**, wherein more than one state machine is associated with the actual thread and each of the state machines that is associated with the actual thread is updated if the actual thread is reset or retired and wherein more than one state machine is associated with the potential thread and each of the state machines that is associated with the potential thread is updated if the potential thread is or is not a good potential thread.

14. The processor of claim **13**, wherein the state machines include simple and correlated predictors and wherein a simple predictor are indexed through at least some of the address bits of the actual thread, thread opportunity, or potential thread of interest and a correlated predictor is indexed through some of the address bits of the actual thread, thread opportunity, or potential thread of interest and bits related to the thread lineage of the actual thread, thread opportunity, or potential thread of interest processed through a function.

15. The processor of claim **14**, wherein the function is an exclusive-OR function.

16. The processor of claim **14**, wherein the simple predictor is not used after the thread lineage is established.

17. The processor of claim **14**, wherein if the potential thread of interest has the same index as a recently retired or

reset actual thread, the final retirement logic does not update the associated state machine in response to whether or not the potential thread of interest is a good potential thread.

18. The processor of claim **14**, wherein the state machine associated with an actual or potential thread may be update twice, once for the potential thread and once for a recently retired actual thread with the same index.

19. The processor of claim **10**, wherein the state machines are indexed through at least some of the address bits of the thread opportunities.

20. A processor comprising:

an execution pipeline to concurrently execute at least portions of actual threads;

detection circuitry to detect speculation errors involving thread dependencies in the execution of the actual threads;

trace buffers outside the execution pipeline to hold instructions of the actual threads;

triggering logic to trigger re-execution of instructions from the trace buffers associated with the speculation errors;

thread management logic to control creation of an actual thread, the thread management logic including reset logic to control whether the actual thread is reset and a thread predictor having state machines to indicate whether thread creation opportunities should be taken or not taken, and wherein if the actual thread is reset, one of the state machines associated with the actual thread is updated in a not take direction; and

final retirement logic to control whether the actual thread is retired, and wherein if the actual thread is retired, the state machine associated with the actual thread is updated in a take direction.

21. The processor of claim **20**, further comprising:

a predictor training mechanism to receive retired instructions and to identify potential threads from the retired instructions and to determine whether a potential thread of interest meets a test of thread goodness, and if the test is met, one of the state machines that is associated with the potential thread of interest is updated in a take direction, and if the test is not met, the state machine associated with the potential thread of interest is updated in a not take direction.

22. The processor of claim **20**, wherein more than one state machine is associated with the actual thread and each of the state machines that is associated with the actual thread is updated if the actual thread is reset or retired.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 6,247,121 B1 Page 1 of 1
DATED : June 12, 2000
INVENTOR(S) : Akkary et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Column 2,
Line 58, delete "**14E**, and **14E**" and insert -- **14E**, and **14F** --.

Column 10,
Line 21, delete "arid" and insert -- and --.

Column 12,
Line 22, delete "riot take state" and insert -- not take state --.

Signed and Sealed this

Seventh Day of May, 2002

Attest:

[illegible]

JAMES E. ROGAN

Attesting Officer *Director of the United States Patent and Trademark Office*

US005317745A

United States Patent [19]

Chan

[11] Patent Number: 5,317,745

[45] Date of Patent: May 31, 1994

[54] MINIMAL INTERRUPT LATENCY SCHEME USING MULTIPLE PROGRAM COUNTERS

[75] Inventor: Stephen H. Chan, Sunnyvale, Calif.

[73] Assignee: Zilog, Inc., Campbell, Calif.

[21] Appl. No.: 818,609

[22] Filed: Jan. 10, 1992

[51] Int. Cl.5 .... G06F 9/00

[52] U.S. Cl. 395/700; 395/775

[58] Field of Search .... 364/DIG. 1, DIG. 2; 395/375, 700, 775, 800

[56] References Cited

U.S. PATENT DOCUMENTS

5,123,094 6/1992 MacDougall 395/375

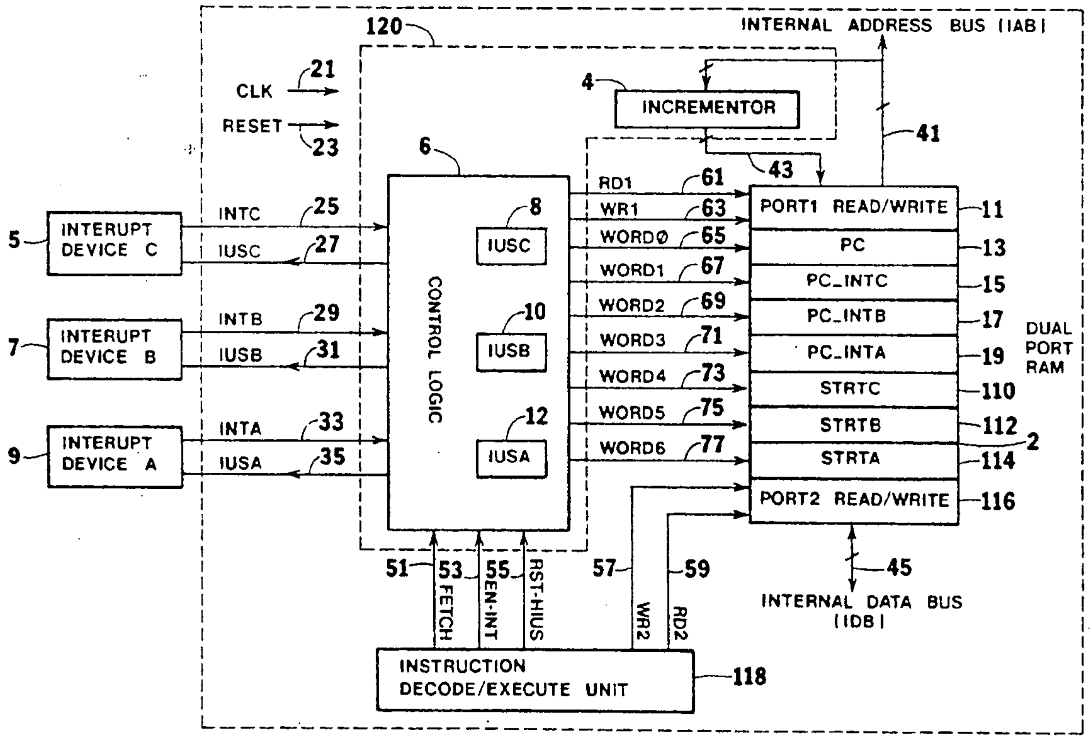
Primary Examiner—Robert L. Richardson

Attorney, Agent, or Firm—Majestic, Parsons, Siebert & Hsue

[57] ABSTRACT

A method and apparatus for using multiple program counters to reduce the latency time of a computer in response to an interrupt or subroutine call using a memory with multiple memory locations for storing the multiple program counters and control means in order to choose which one of the memory locations is used as a current program counter. Additionally, the use of a memory location to store the starting address of the interrupt subroutine is also disclosed.

14 Claims, 6 Drawing Sheets



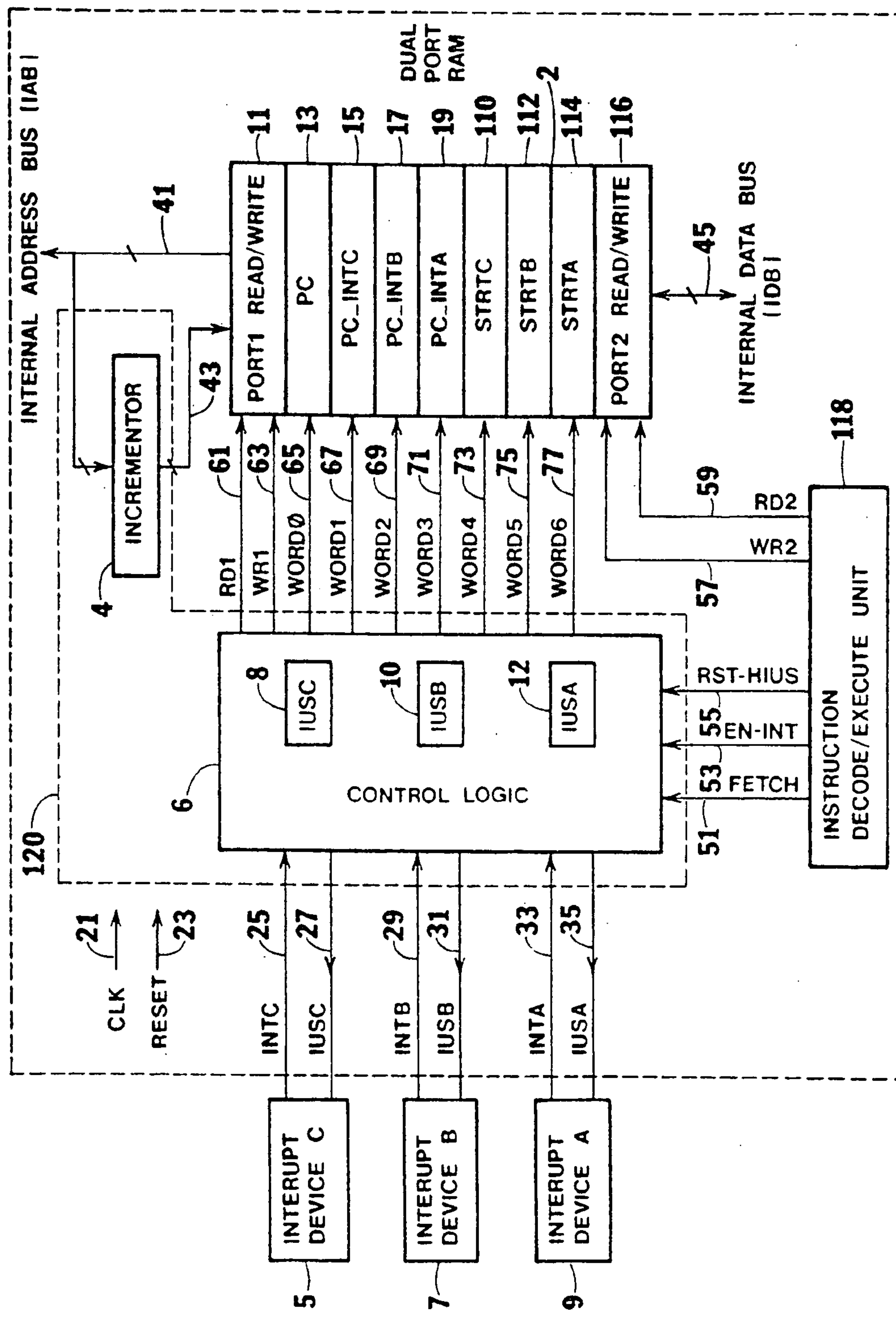
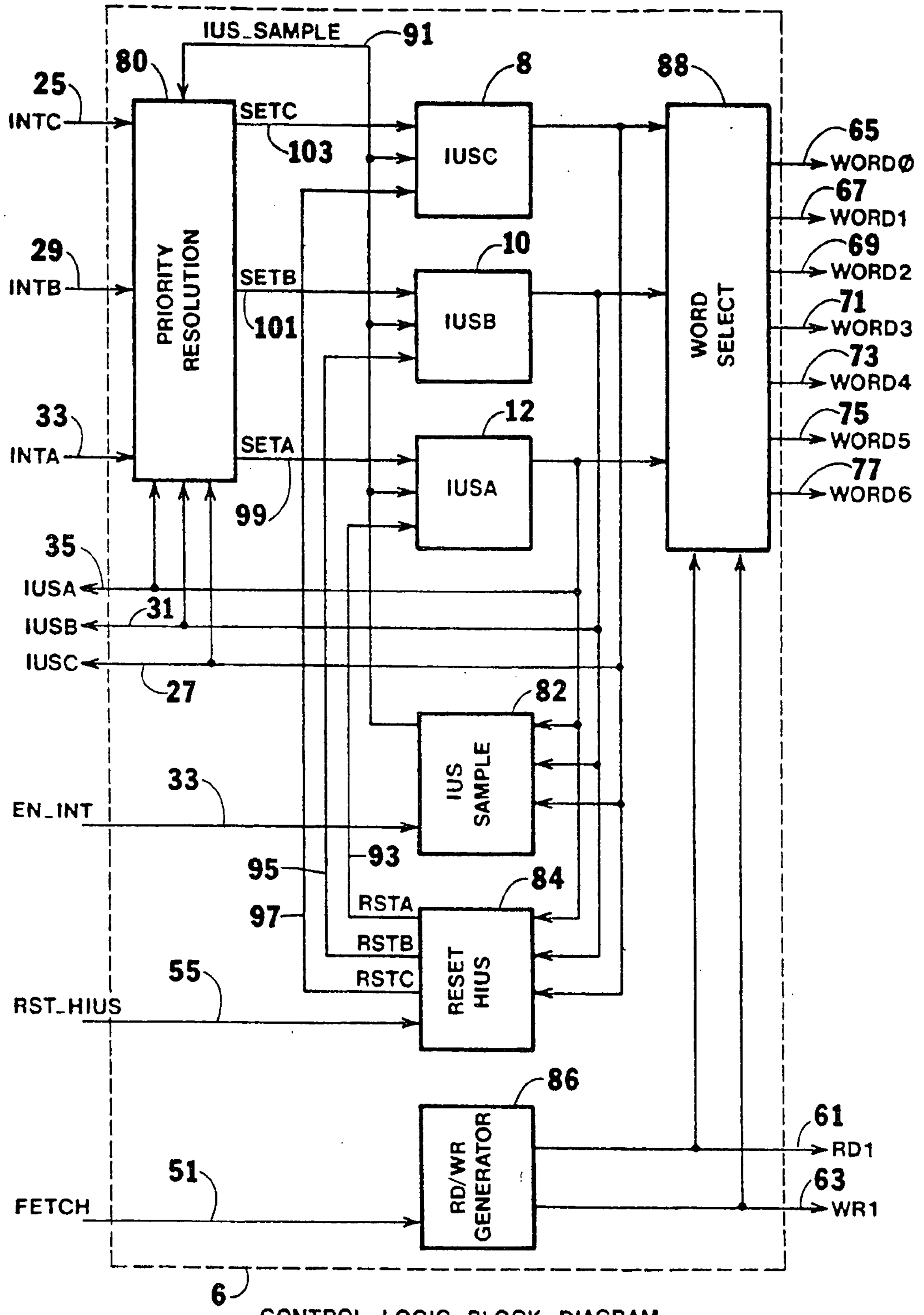
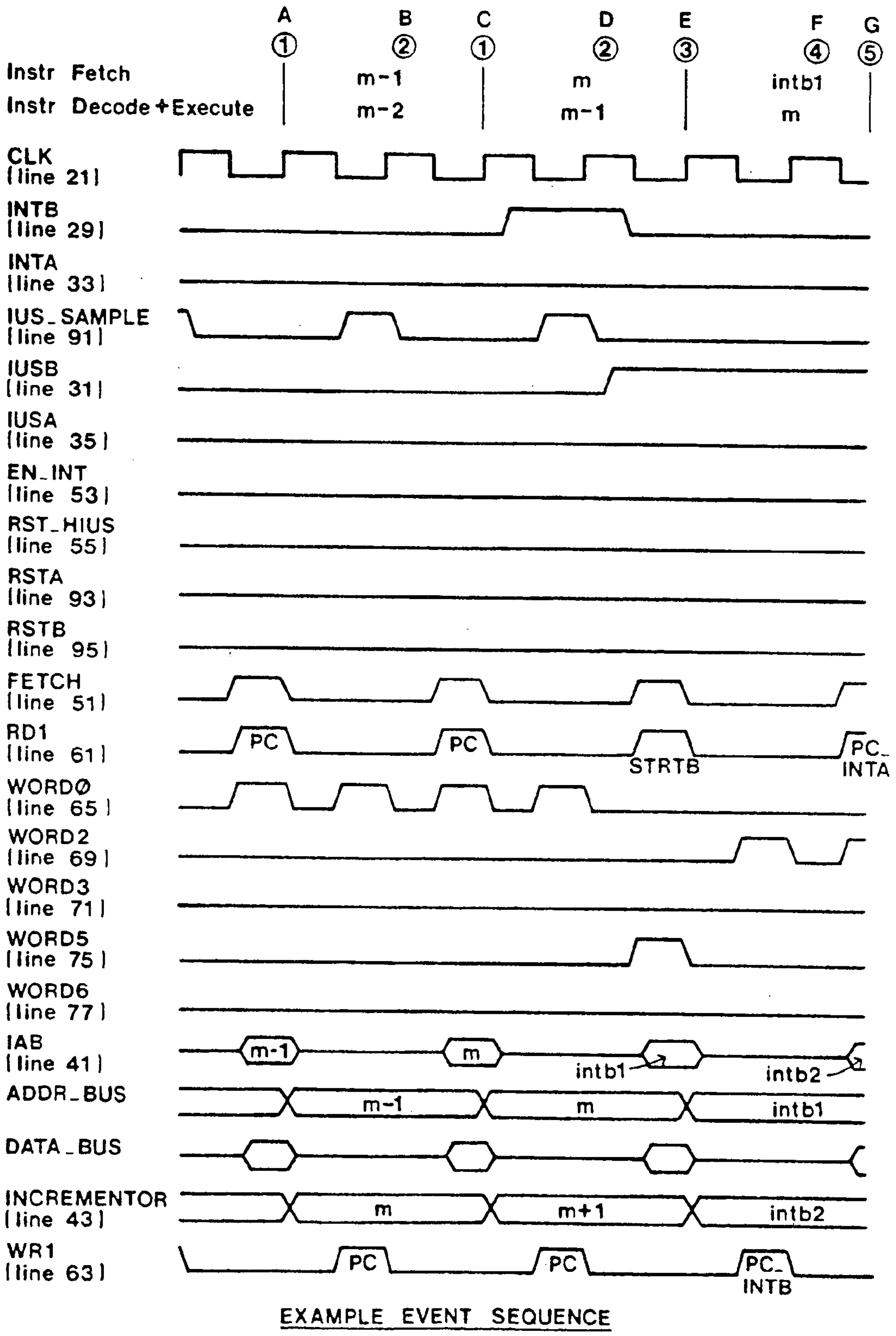

INTERNAL ADDRESS BUS (IAB)
41
43
45
INCREMENTOR
4
120
DUAL PORT RAM
2
PORT1 READ/WRITE
PC
PC_INTC
PC_INTB
PC_INTA
STRTC
STRTB
STRTA
PORT2 READ/WRITE
11
13
15
17
19
110
112
114
116
INTERNAL DATA BUS (IDB)
RD1
WR1
WORDØ
WORD1
WORD2
WORD3
WORD4
WORD5
WORD6
61
63
65
67
69
71
73
75
77
CONTROL LOGIC
6
IUSC
IUSB
IUSA
8
10
12
RST-HIUS
EN-INT
FETCH
51
53
55
57
59
RD2
WR2
INSTRUCTION DECODE/EXECUTE UNIT
118
CLK
RESET
21
23
INTC
IUSC
INTB
IUSB
INTA
IUSA
25
27
29
31
33
35
INTERUPT DEVICE C
INTERUPT DEVICE B
INTERUPT DEVICE A
5
7
9

fig._1.



CONTROL LOGIC BLOCK DIAGRAM

fig._2.



EXAMPLE EVENT SEQUENCE

fig._3A.

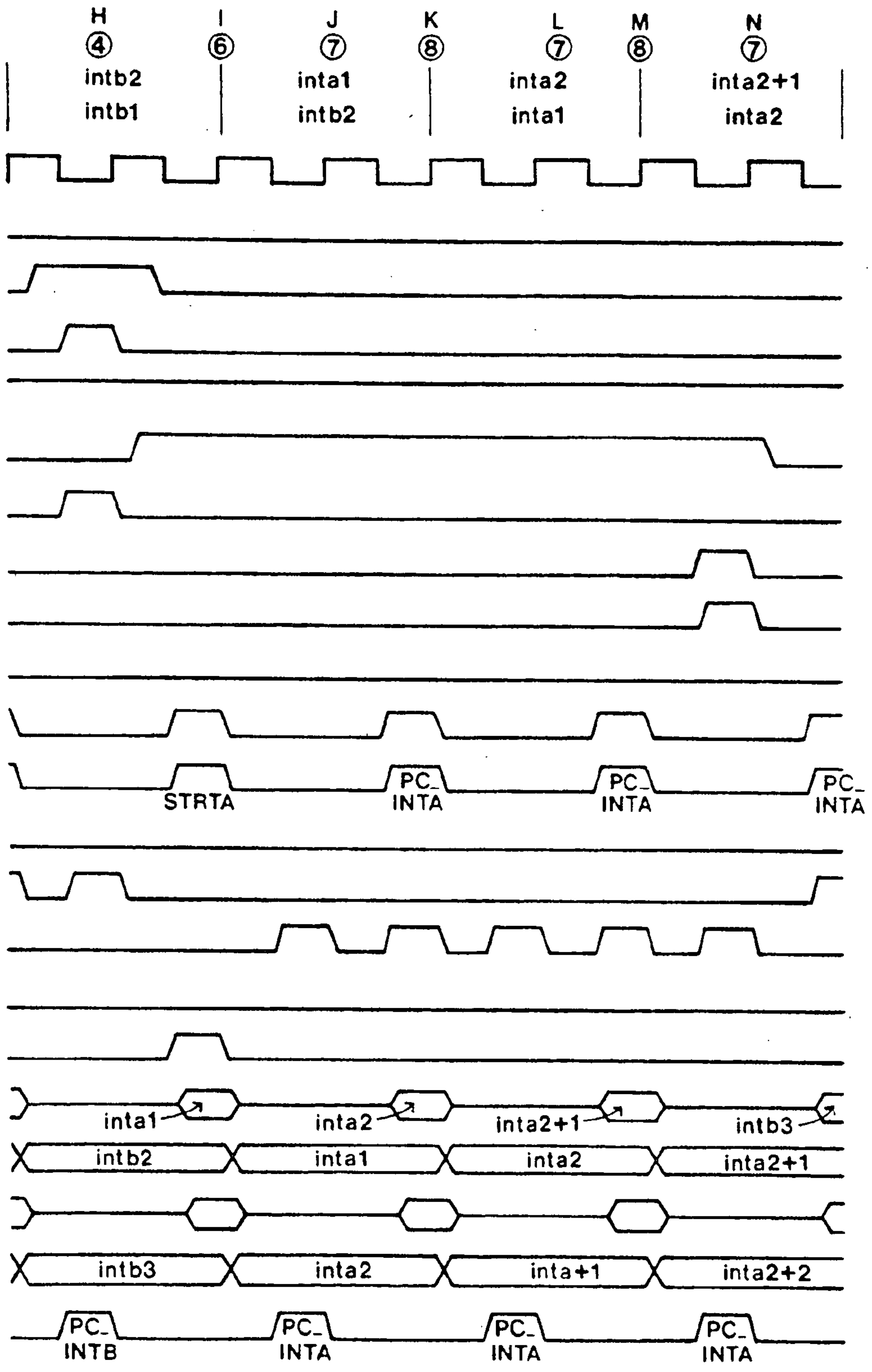


fig._3B. (place to right of fig._3A.)

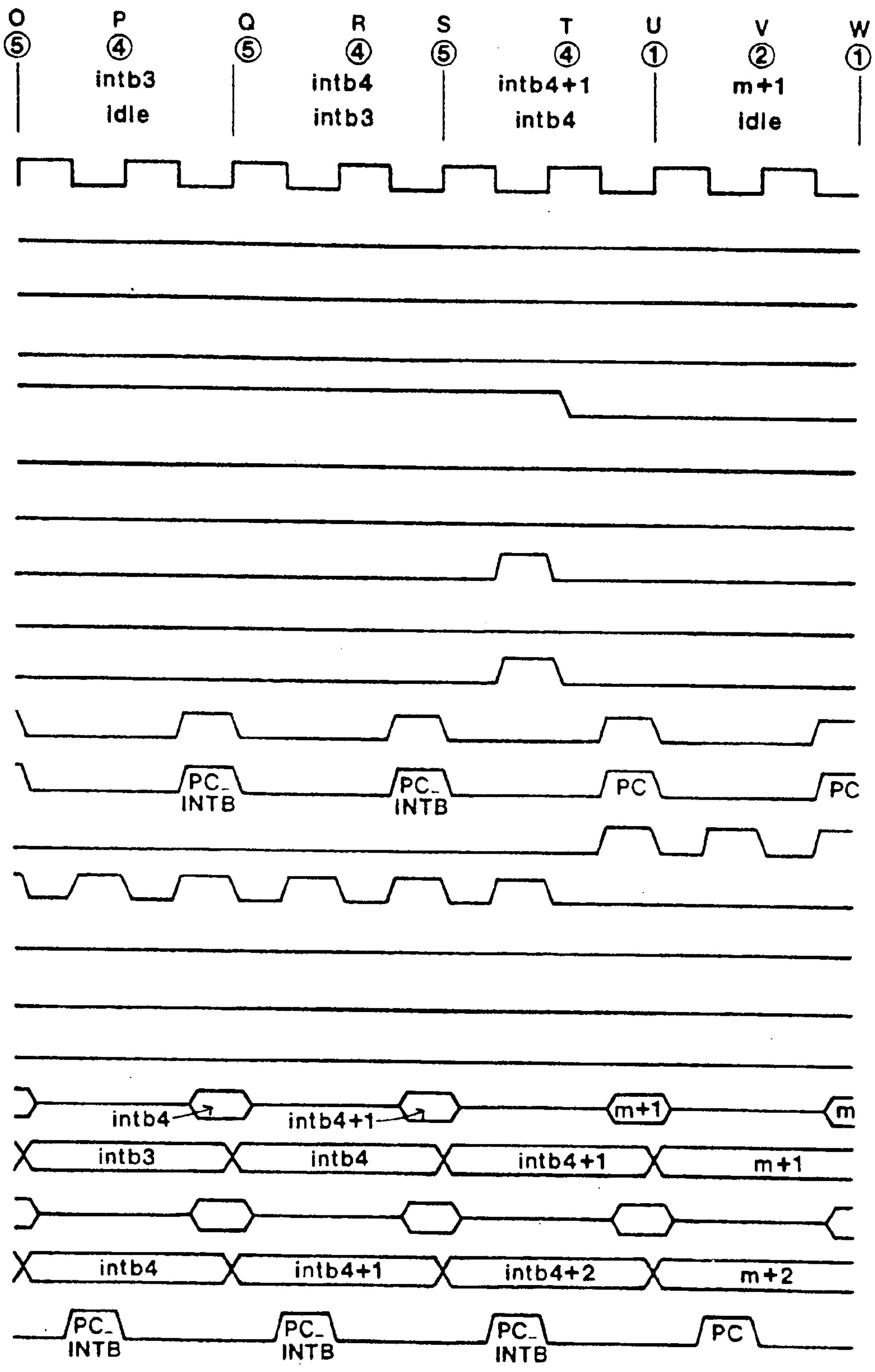


fig._3C. (place to right of fig._3B.)

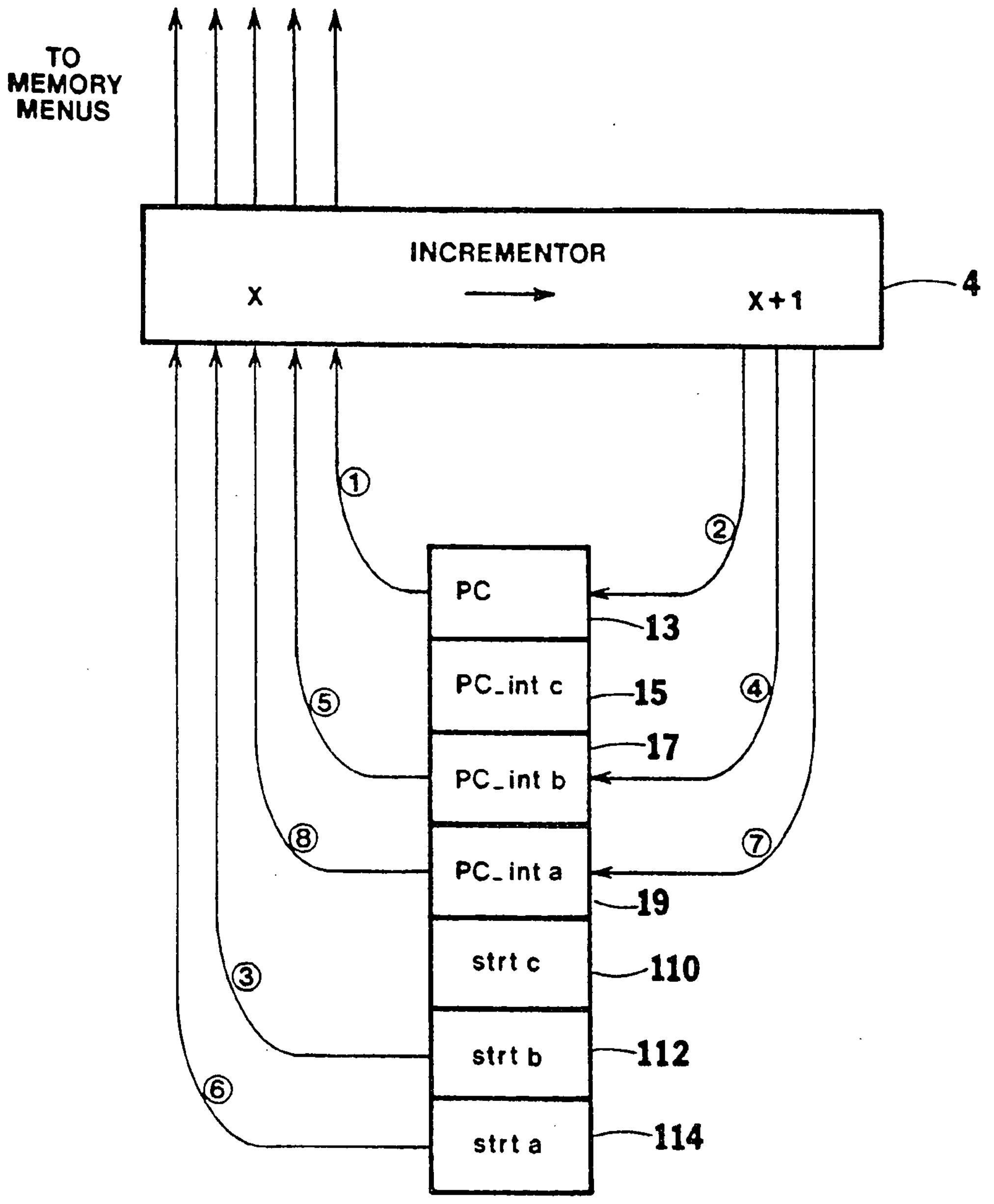

TO MEMORY MENUS
INCREMENTOR
X
X+1
4
PC
13
PC_int c
15
PC_int b
17
PC_int a
19
strt c
110
strt b
112
strt a
114
1
2
3
4
5
6
7
8

fig._4.

MINIMAL INTERRUPT LATENCY SCHEME USING MULTIPLE PROGRAM COUNTERS

Background of the Invention

This application concerns the method and apparatus for reducing the interrupt latency time.

By common design practice, a microprocessor central processing unit (cpu) samples the presence of an interrupt request close to the end of the last transaction of the instruction being executed. The cpu saves the interrupted program's next instruction address, currently residing in its program counter (pc), and then loads the starting address of the interrupt service routine into the pc. The new pc value then goes to the address bus, so that the first instruction of the service routine can be fetched for execution.

The interrupted program address is generally saved by "pushing" it into a memory stack, which is a block of memory utilized as a first-in-last-out buffer, or by storing the address in a temporary register.

The sampling of the presence of an interrupt request, the saving of the current pc value, the loading of the service routine pc value, up to the start of the first fetch transaction for the service routine, require a certain amount of time. This time, sometimes expressed as a number of cpu clock periods, is defined as the interrupt latency.

In systems where multiple interrupt request sources are present, priority resolution has to be performed so that the cpu can determine which interrupt request to service. Depending on the system architecture, this priority resolution may or may not add to the interrupt latency.

At the end of an interrupt service routine, the saved program address is reloaded into the pc, so that the interrupted program can resume its execution.

A microprocessor manufactured by Zilog, Inc., the Z80180, can be cited as a reference example. The Z80180 samples its interrupt request 0 input at the cpu clock's falling edge that is 1.5 clock periods earlier than the end of the last transaction of an instruction being executed. In interrupt mode 1, if the request input is active, the Z80180 would next perform an interrupt acknowledge transaction, followed by pushing its current pc value to memory stack, done with two memory write transactions, before the first instruction of the interrupt service routine is fetched from a fixed memory location of hexadecimal 0038. Counting from the above mentioned cpu clock's falling edge, the interrupt latency is 12.5 clock periods.

At the end of the interrupt service routine, the Z80180 reloads its saved pc value with two memory read transactions, accomplished in six clock periods. This calculation does not include the time required to fetch the "return from interrupt" instruction that invokes the pc value recovery.

Additionally, in prior computer systems on a response to a subroutine call, the pc value and other values are stored in a stack in a manner similar to the interrupt service as described above. This also creates a time delay in the computer processing.

It is an object of the present invention to have an interrupt system with a reduced latency time for responding to an interrupt.

An additional object of the present invention is to reduce the latency time of a computer in response to a subroutine call.

SUMMARY OF THE INVENTION

In accordance with the principles of the present invention, the above and other objectives are realized by utilizing a program counter apparatus for reducing the latency time of a computer in response to an interrupt or subroutine call where the apparatus has a memory with a plurality of memory locations which can correspond to program counters, where one of the plurality of memory locations corresponds to the general program counter and a number of other of the memory locations correspond to alternate program counters, and where the apparatus has control means connected to the memory for controlling which one of the memory locations is used by the computer as the current program counter.

Additionally, another aspect of the invention is directed to a method for reducing the latency time of a computer in response to interrupts from interrupt devices. This method uses a computer having a memory with memory locations, where the method has the steps of establishing a memory location in the memory corresponding to the general program counter and memory locations in the memory corresponding to alternate program counters, prioritizing the interrupts received from interrupt devices to choose an interrupt device and associated subroutine to be serviced, switching the memory location used in the computer as a program counter from the memory location corresponding to the general program counter to the memory location corresponding to the alternate program counter associated with the chosen interrupt device, executing an interrupt subroutine and then switching back to the memory location used by the computer as the program counter back from the memory location corresponding to the alternate program counter associated with the chosen interrupt device to the memory location corresponding to the general program counter.

The use of multiple program counters reduces the interrupt latency time because the old program counter value does not have to be stored into a stack and then retrieved from a stack, but the computer can merely switch from using one program counter, for example, the general program counter, to another program counter, for example, an alternate program counter associated with an interrupt device. When the subroutine of the interrupt device is finished, the computer can switch from the program counter associated with the interrupt device back to the general program counter.

BRIEF DESCRIPTION OF THE DRAWINGS

The above and other features and aspects of the present invention will become more apparent upon reading the following detailed description in conjunction with the accompanying drawings, in which:

FIG. 1 is a schematic diagram showing the control means (control logic unit and the incrementor), the multiple program counter memory, the interrupt devices, and the instruction decode execute unit;

FIG. 2 is a schematic diagram of the control logic unit;

FIG. 3A is a timing diagram for the apparatus shown in FIGS. 1 and 2;

FIG. 3B is a continuation of the timing diagram in FIG. 3A;

FIG. 3C is a continuation of the timing diagram in FIG. 3B; and

FIG. 4 is a diagram showing the steps of switching between the program counters.

DETAILED DESCRIPTION

FIG. 1 is a schematic diagram of one embodiment of the present invention. Instead of using a traditional program counter, which is a reloadable upcounter, multiple program counters are stored in a memory 2. The memory 2 has memory locations corresponding to multiple program counters, the general program counter 13, and alternate program counters 15, 17, 19. PC_INTC 15 is an alternate program counter associated with the interrupt device C 5. PC_INTB 17 is an alternate program counter corresponding to the interrupt device B 7. PC_INTA 19 is an alternate program counter corresponding to the interrupt device A 9. The apparatus of the present invention could, of course, have more alternate program counters than interrupt devices, so that the apparatus would be able to add additional interrupt devices. In addition, the present invention could be used with as many interrupt devices as the system designer desires. In one embodiment of the present invention, the memory is a dual port RAM (random access memory).

STRTC 110, STRTB 112 and STRTA 114 are memory locations in the memory 2 that correspond to the starting address of the interrupt subroutines that service the associated interrupt devices.

A control means 120 controls the use of the program counters in the memory 2. The control means 120 consists of a control logic unit 6 and an incrementor 4. The selection of the memory location in the memory 2 used as the current program counter is performed by control logic unit 6.

The control logic unit 6 also samples the interrupt signals from the interrupt devices 5, 7, and 9. Interrupt signal INTC from the interrupt device C 5 is sent to the control logic unit 6 on line 25. Interrupt signal INTB is sent to the control logic unit 6 over line 29 from interrupt device B. Interrupt signal INTA is sent to the control logic unit over line 33 by the interrupt device A. These three interrupt signals have implicit priority assignments corresponding to the priority of the interrupt devices. For example, INTA has the highest priority, INTB has the second highest priority, and INTC has the lowest priority. At close to the end of the last transaction of the instruction being executed, the control logic unit captures the logic states of these asynchronous inputs and decides which of the active interrupt signals has the highest priority assignment. It then sets one of three interrupt under service (ius) flags to reflect this resolution. Once an ius flag is set, it inhibits any lower priority interrupt signals from being recognized. The interrupt under service flags can be output from the cpu as status indicators, and if so desired, the outputs can be properly timed to align their changes to the start of the next transaction. The control logic unit 6 sends the status indicator IUSA over line 35 to interrupt device A, the status indicator IUSB over line 31 to interrupt device B, and the status indicator IUSC over line 27 to interrupt device C. The interrupt devices can be designed to recognize the active going edges of the appropriate status indicators to deassert their interrupt request signals.

In addition, interrupt nesting is allowed. By common design practice, many systems automatically disable

request input sampling when entering an interrupt service subroutine. The service routine can then enable the interrupt request sampling again. The enabling of the interrupt request sampling is generally done by executing an enable interrupt instruction in the interrupt service subroutine. If an enable interrupt instruction is executed in an first interrupt service subroutine and a second interrupt request of higher priority occurs, it is possible to have more than one ius flag in the active state. This is because the corresponding active ius flag does not inhibit request inputs from higher priority interrupt devices from being sampled.

Generally, the memory location in memory 2 corresponding to the highest priority ius flag in effect is read to or written from through the port 1 11. That is, the highest priority active ius flag dictates that the corresponding alternate program counter be read to the internal address bus, and the incremented value written back into the same memory location.

If the IUSC flag is the highest priority ius flag set, then a read (RD1) from the control logic unit 6 to the port 1 11 of memory 2 over line 61 would cause the word contained in PC_INTC 15 to be sent across the internal address bus 41. In the same step, the word from PC_INTC 15 is sent to the incrementor 4. The incrementor 4 automatically stores the value sent up the internal address bus 41 and increments this value so that a write operation can use the incremented value. A write operation (WR1) over line 63 from the control logic unit 6 to the port 1 11 over line 63 while the IUSC flag is the highest priority ius flag set would cause the incremented value from the incrementor 4 to be written over bus line 43 into PC_INTC 15. PC_INTC 15, the alternate program counter associated with interrupt device C, is used as the current program counter while IUSC is the highest priority ius flag set.

If no interrupt under service flag is set, then a RD1 from the control logic unit 6 across line 61 to port 1 of the memory 2 will cause the contents of the general program counter pc 13 to be sent across the internal address bus 41 and to the incrementor 4. A WR1 signal across line 63 from the control logic unit 6 to port 1 11 of the memory 2 will cause the incremented value from the incrementor 4 to be placed into the memory location corresponding to the general program counter pc 13.

A special reset highest priority active ius flag instruction (RST_HIUS) is used as a last instruction in interrupt service routine. After this instruction is executed, the memory location of the alternate program counter corresponding to the next highest priority active ius flag would then be used as the current program counter.

An exception to the word selection rule occurs to get the first address of the interrupt subroutine. For the first iteration, when the ius flag goes active, the corresponding interrupt service routine starting address is read out of memory locations 112, 114 or 116 and the incremented value is written back to the corresponding alternate program counter memory location 15, 17 and 19. In this way, the service routine starting address is preserved for future use. For example, the first time an ius flag such as IUSC is set, the subroutine starting address memory location for the interrupt subroutine associated with the interrupt device C stored in STRTC 110 is sent up the internal address bus 41 and sent to the incrementor 4. Then, when the write signal (WR1) is sent, the incremented starting address value is sent across bus 43 back to PC_INTC 15. Since the contents of the subroutine starting address memory location for interrupt device C contained in STRTC 110 is not written upon, the subroutine starting address is saved.

Additionally, as discussed above, when none of the ius flags are active, the general program counter 13 is selected. Words 0 through 6, on lines 65, 67, 69, 71, 73, 75 and 77 send a signal to a memory location in memory 2 that controls which memory location is to be read from or written to. For example, the first time an ius flag such as IUSC 8 is set, a signal is sent across word 4 on line 73 and RD1 on line 61 to cause the contents of STRTC 110 to be read and sent across the internal address bus 41 and to the incrementor 4. A signal is sent across word 1, line 67 to PC_INTC 15 and WR1, line 63 to cause the incremented value from the incrementor 4 across the bus 43 to be written into PC_INTC 15, the memory location corresponding to the alternate program counter associated with the interrupt device C. In this manner, the word signals, the RD1 signal and the WR1 signal, the reading and writing of the memory locations in the memory 2 is controlled.

Port 2 116 of the memory 2 is connected to the internal data bus. Under program control, the interrupt service routine starting addresses, which are stored STRTC 110, STRTB 112, and STRTA 114, can be written through port 2 116. At certain instruction executions, the contents of the memory locations of the memory 2 can be modified. With properly designed system timing, the two ports 11 and 116 of memory 2 should not have access conflicts. The reading and writing through port 2 116 from the instruction decode/execute unit 118 is done with a WR2 over line 57 and a RD2 over line 59. For the balance of this discussion, it is assumed that interrupt service routines' starting addresses have been written to STRTA, STRTB, and STRTC.

A system reset on line 23 forces the general program counter pc 13 to 0 and the IUSA, IUSB and IUSC flags to inactive.

Examples of interrupt devices used in the present invention include such peripherals such as a serial communications controller (SCC), a counter/timer controller (CTC), or a parallel input/output port (PIO).

FIG. 2 is a schematic diagram of the control logic unit 6. The IUS SAMPLE block 82 in the control logic unit 6 initiates the generation of IUS_SAMPLE pulses across line 91, when an EN_INT signal is sent from the instruction decode/execute unit 118, shown in FIG. 1, across line 53. At the rising edge of an ius signal IUSC, IUSB, or IUSA, the IUS SAMPLE block stops generating IUS SAMPLE pulses across line 91. Generation of IUS_SAMPLE pulses across line 91 can resume with the execution of another enable interrupt instruction, which activates EN_INT across line 53 from the instruction decode/execute unit 118 shown in FIG. 1.

At the rising edge of the IUS_SAMPLE signal across line 91 to the priority resolution block 80, the state of the external interrupt request INTC across line 25 from interrupt device C 5 shown in FIG. 1, INTB across line 29 from interrupt device B 7 shown in FIG. 1, and INTA across line 33 from interrupt device A 9 are has the highest priority of the interrupt requests or is of a higher priority than any active ius flag, a signal is sent to the corresponding ius block, by SETC line 103, SETB line 101 or SETA line 99, to set the corresponding ius flags.

Note that if IUS_SAMPLE is not active, none of the SETC or SETB or SETA signals can go active.

A signal across the SETC line 103, SETB line 101, or SETA line 99 goes to the corresponding ius set flag blocks IUSC 8, IUSB 10, IUSA 12, and the corresponding ius flag goes active at the following edge of the IUS_SAMPLE.

When the instruction decode/execute unit 118 shown in FIG. 1 executes a reset highest priority active ius flag instruction, a RST_HIUS signal is sent across line 55. The RESET HIUS block 84 sends a reset signal RSTC across line 97, RSTB across line 95 or RSTA across line 93, to reset the corresponding ius flag in block IUSA 8, IUSB 10, or IUSC 12 on the reset signal's following edge based on which ius flag is active and has the highest priority.

Each FETCH signal across line 51 from the instruction decode/execute unit 118, shown in FIG. 1, causes the RD/WR generator 86 to generate a RD1 pulse across line 61. Each RD1 pulse across line 61 is followed by a WR1 pulse across line 63 also generated by the RD/WR generator 86. That is, one FETCH signal causes both a RD1 and a WR1 pulse.

For each RD1 or WR1 pulse, the word select block 88 causes a word pulse to be sent to a memory location. That is, word 0 over line 65, word 1 over line 67, word 2 over line 69, word 3 over line 71, word 4 over line 73, word 5 over line 75 or word 6 over line 77 is selected by the word select block 88. The word select block 88 samples the ius flags to determine which word to select. As discussed above, the general rule to determine which word is selected active is to select the program counter memory location associated with the highest priority active ius flag. For example, if IUSB is active and has the highest priority, the word select block selects word 2 and the alternate program counter associated with interrupt device B is used as the current program counter. Two exceptions are word 0 which is enabled to write and read from the general program counter memory location 13 when none of the ius flags are active, and words 4, 5, and 6 which are set at the rising edge of the ius flag to get the starting address of the corresponding interrupt service routine. Word 4, word 5 or word 6 enable the corresponding starting address location stored in STRTC 110, STRB 112 or STRA 114, containing the subroutine starting address of the interrupt device being serviced, to be output. STRTC 110, STRB 112 or STRA 114 are accessed only once at the beginning of each interrupt subroutine.

FIGS. 3A, 3B and 3C show an example of an event sequence. For example, looking at FIG. 3A, at time A, the RD1 across line 61 and the word 0 across line 65 are set so that the memory location of the general program counter 13 of the memory 2, shown in FIG. 1, is read across the internal address bus 41 and sent to the incrementor 4. The contents of the internal address bus can be used as the address of the instruction to be executed. The contents of the internal address bus across line 41 is $m-1$. This value is sent to the incrementor 4. It is then incremented from $m-1$ to m. At time B, since the word 0 line 65 and the WR1 over line 63 are both set, the incremented value from the incrementor 4, (m), is written into the memory location 13 corresponding to the general program counter.

At time C, since the RD1 and word 0 are set, the contents of the memory location 13 (m) is sent across the internal address bus and to the incrementor 4. Soon afterward, an INTB signal from the interrupt device B 7 is sent to the control logic unit 6, and since the

IUS_SAMPLE signal is pulsing and no higher priority ius flag is active, the IUSB flag is set.

Next, at time E, when the RD1 flag and the word 5 flag are set, the interrupt subroutine starting address corresponding to interrupt device B which is stored in STRB 112 is sent across the internal address bus and to the incrementor 4. The incrementor 4 increments this value (intb1) to the value intb2. At time F, word 2 across line 69 and the WR1 signal across line 63 are set, so that the value intb2 is written to PC_INTB 17, the memory location corresponding to the alternate program counter associated with the interrupt device B. In this manner, the starting location of the subroutine is sent out across the internal address bus, incremented, and stored back into the alternate program counter associated with the interrupt device B.

FIG. 4 is a diagram of the steps shown in FIGS. 3A, 3B and 3C. At time A, step 1 occurs; Step 1 is a read from the general program counter 13. The contents of the general program counter is sent across the internal address bus and to the incrementor 4. At time B, Step 2 occurs, which is the writing of the incremented value to general program counter 13. At time C, step 1 is repeated. At time D, step 2 is repeated. When an interrupt occurs, the contents of the memory location corresponding to the general program counter now contains the address of the next instruction to be executed after a return from the interrupt.

Before time E, an INTB interrupt signal is sent by the interrupt device B to the control logic unit 6. The control logic unit 6 sets the IUSB flag at the down edge of the IUS-SAMPLE signal. The interrupt device B is now the chosen interrupt device.

At time E, step 3 occurs; step 3 is a read from the memory location 112 which corresponds to the starting address of the interrupt subroutine associated with the interrupt device B. This starting address is sent across the internal address bus and to the incrementor 4, which increments the value. Next, at time F, step 4 occurs; step 4 is a write of the incremented value to PC_INTB 17, the memory location corresponding to the alternate program counter associated with the interrupt device B. At time G, step 5 occurs, which is a read from the PC_INTB 17, the contents of which are sent across the internal address bus and to the incrementor 4.

Note that at time F and time G an IUS_SAMPLE signal does not occur because the subroutine for the interrupt device B has not enabled the sampling of the other interrupt devices. When the interrupt subroutine for the first chosen interrupt device (interrupt device B) enables the sampling of the other interrupt devices, the IUS_SAMPLE signal resumes at time H. A subroutine corresponding to an interrupt device need not allow the sampling of the other interrupt devices, and such a subroutine would not be interrupted by other higher priority interrupt devices.

At time H, step 4 repeats so that PC_INTB 17 now contains intb3; when the next interrupt subroutine is returned from, the alternate program counter corresponding to the interrupt device B contains the memory location of the next instruction of the interrupt subroutine for interrupt device B. During time H, the INTA signal is sampled because INT_SAMPLE is active. Since INTA is of a higher priority than IUSB, IUSA is set. Note that both IUSB and IUSA are now set. After the next write step occurs, step 4 at time H, the second chosen interrupt device may be serviced.

At time I, step 6 occurs; step 6 is a read of the starting address of the subroutine A from STRTA 114 to the internal address bus and the incrementor 4. At time J, step 7 occurs; step 7 is a write of the incremented value to the PC_INTA 19, which is the memory location corresponding to the alternate program counter associated with the interrupt device A. At time K, step 8 occurs; step 8 is a read from PC_INTA 19 across the internal address bus and to the incrementor 4.

Skipping to time 0, the interrupt subroutine A has finished, so RST_HIUS is set and IUSA is cleared. Since IUSB remains set, the interrupt subroutine for interrupt device B can resume by repeating step 5 without needing to restore any value to the alternate program counter, PC_INTB 17. At time P, step 4 repeats.

The alternate program counter corresponding to interrupt device B (PC_INTB 17) is used as the current program counter for a few more steps until time U. When the interrupt subroutine for the interrupt device B is finished, RST_HIUS is set, IUSB is reset and the general program counter is used as the current program counter.

The present invention obviates the steps of saving the current pc value and loading the new pc value when entering a service subroutine, thereby reducing interrupt latency. Saved pc value recovery at the end of a service routine is eliminated. The scheme is readily adaptable to the needs of various cpu architectures, including considerations for the instruction sets and transaction protocol timing.

Additionally, the multiple pc scheme can be used for regular subroutine calls when non-reentrant programming is used. Non-reentrant programs are programs where a subroutine does not call itself. When non-reentrant programming is used, the present scheme can reduce the latency time of switching to a called subroutine.

Various details of the implementation and the method are merely illustrative of the invention. It will be understood that various changes in such details may be within the scope of the invention, which is to be limited only by the appended claims.

What is claimed is:

1. A program counter apparatus for reducing the latency time of a computer in response to an interrupt, said apparatus comprising:

a memory having memory locations, wherein a general program counter value is located at one of said memory locations and alternate program counter values are located at a number of other of said memory locations; and

control means connected to said memory for controlling which one of said memory locations contains the current program counter value.

2. The apparatus in claim 1, wherein said control means includes a control logic unit and an incrementor, said incrementor connected to said memory and said control logic unit, wherein said incrementor increments the current program counter value, and wherein said control logic unit controls the incrementor.

3. (New) The apparatus of claim 2, wherein said memory includes at least one interrupt subroutine starting address memory location that stores an interrupt subroutine starting address and said control means includes means for putting the interrupt subroutine starting address from said interrupt subroutine starting address memory location to the incrementor to be incre-

mented and then placing the incremented value into one of said alternate program counter value locations.

4. The apparatus in claim 1, wherein said control means is connected to a number of interrupt devices which are designed to send interrupt signals to the control means and wherein said control means includes means for using said interrupt signals to select which memory location contains the current program counter value.

5. A method for reducing the latency time of a computer in response to interrupts from interrupt devices, said computer including a memory with a plurality of memory locations, said method comprising the steps of:

(a) establishing a general program counter value located at a first of said memory locations in said memory, establishing alternate program counter values associated with the interrupt devices, said alternate program counter values located at other memory locations in said memory and setting the location of the current program counter value as the first memory location containing the general program counter value;

(b) prioritizing the interrupts received from the interrupt devices to choose one of the interrupt devices to be serviced by executing its associated subroutine;

(c) a switching the location of the current program counter value from the first memory location containing the general program counter value to another memory location containing the alternate program counter value associated with the chosen interrupt device;

(d) thereafter, executing the interrupt subroutine associated with the chosen interrupt device; and

(e) subsequently, switching the location of the current program counter value back from the another memory location containing the alternate program counter value associated with the chosen interrupt device to the first memory location containing the general program counter.

6. The method of claim 5, wherein step (c) includes the additional step of getting an interrupt subroutine starting address from the memory, incrementing this starting address, and placing the incremented value in the another memory location to be used as the alternate program counter value associated with the chosen interrupt device.

7. The method of claim 5, said method further comprising the following step after step (c) but before step (e):

determining whether to service a second interrupt device including prioritizing the interrupts received from interrupt devices and comparing them to the chosen interrupt device.

8. The method of claim 7, wherein the additional step of determining whether to service a second interrupt device includes the step of not allowing a service to the second interrupt device unless the interrupt subroutine associated with the chosen interrupt device enables the service of the second interrupt device and the second interrupt device has a greater priority than the chosen interrupt device.

9. The method of claim 7, further comprising the following steps in order after the determining step:

switching the location of the current program counter value from the another memory location containing the alternate program counter value associated with the chosen interrupt device to yet another memory location containing a second alternate program counter value associated with the chosen second interrupt device;

executing the interrupt subroutine associated with the chosen second interrupt device;

switching the location of the current program counter value back from the yet another memory location containing the second alternate program counter value associated with the chosen second interrupt device to the another memory location containing the alternate program counter value associated with the chosen interrupt device.

10. A program counter apparatus for reducing the latency time of a computer in response to a subroutine call, said apparatus comprising:

a memory having memory locations, wherein a general program counter value is located at one of said memory locations and alternate program counter values are located at a number of other of said memory locations; and

control means connected to said memory for controlling which one of said memory locations contains the current program counter value.

11. The apparatus in claim 10, wherein said control means includes means for selecting one of said alternate program counter values to be used as the current program counter value in response to a subroutine call from an Instruction Decode/Execute unit and means for selecting the general program counter value to be used as the current program counter value when said Instruction Decode/Execute unit finishes executing the subroutine.

12. The apparatus in claim 10, wherein said control means includes a control logic unit and an incrementor, said incrementor connected to said memory and said control logic unit, wherein said incrementor increments the current program counter value, and wherein said control logic unit controls the incrementor.

13. The apparatus of claim 12, wherein said memory includes at least one subroutine starting address memory location that stores a subroutine starting address and said control means includes means for putting the subroutine starting address from said subroutine starting address memory location to the incrementor to be incremented and then placing the incremented value into one of said alternate program counter value locations.

14. The apparatus of claim 13, wherein said memory includes a first port connected to said control means and a second port connected to an Instruction Decode/Execute unit.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO.	:	5,317,745
DATED	:	May 31, 1994
INVENTOR(S)	:	Stephen H. Chan

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

In Column 8, line 62:
replace "(New) The apparatus of claim 2, wherein said" with:

--The apparatus of claim 2, wherein said--

In Column 9, line 27:
replace "(c) a switching the location of the current program" with:

--(c) switching the location of the current program--

Signed and Sealed this

Twenty-fifth Day of October, 1994

Attest:

[illegible]

BRUCE LEHMAN

Attesting Officer

Commissioner of Patents and Trademarks